Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master Thesis

Design and Implementation of an Evaluation Testbed for Fog Computing Infrastructure and Applications

Leon Graser

Course of Study:	Computer Science
Examiner:	Prof. Dr. Dr. Kurt Rothermel
Supervisor:	Dipl.-Inf. Ruben Mayer
Commenced:	April 25, 2017
Completed:	September 26, 2017
CR-Classification:	C.2.1, C.2.4, C.4

Abstract

Besides the popular Cloud Computing paradigm, a new approach to distributed computation, known as Fog Computing, has been emerging in the last few years. This approach suggests, that the intelligence should move from the data centers to the network level. In the past years, Fog Computing has been gaining more attention, which has led to the rise in projects and publications. Unfortunately, there is very little support to test and evaluate Fog Computing applications. Aside from expensive real world deployments, there are few tools to simulate the behavior. Since simulation does not execute the application to be tested, the results are less accurate than in an emulated environment. Emulation offers a trade-off between evaluation costs and accurate results. This work proposes a new approach to read in network topologies from different sources and uses them to evaluate user defined Fog Computing applications. To identify the edge of those networks an algorithm is presented. Also, a heuristic to place fog nodes cost optimal within a user defined proximity of the edge is suggested. The final outcome can be exported to a network emulator like MaxiNet in combination with Docker. This approach is implemented in EmuFog and published open source. It is easily extensible for future use, platform-independent, and flexible for different applications to test. A user can specify the computing capabilities (i.e.RAM) of each node type and define the associated Docker image to run. Hierarchies can be built using dependencies between fog node types. Also, an evaluation is carried out to measure the algorithms presented. For the edge identification and the fog node placement, the evaluation shows reasonable running times even for bigger network sizes of up to $10,000$ nodes. In the evaluated networks the heuristic shows an average deviation of 1.2, and in the worst-case scenario, a deviation of $\frac{5}{3}$ of the cost optimal result.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

List of Acronyms

AS Autonomous System

AWS Amazon Web Services

BFS Breadth First Search

BGP Border Gateway Protocol

BRITE Boston Representative Internet Topology Generator

CAIDA Center of Applied Internet Data Analysis

CORE Common Open Research Emulator

CPU Central Processing Unit

DFN Deutsches Forschungsnetz

GLP Generalized Linear Preference

GRE Generic Routing Encapsulation

GT-ITM Georgia Tech Internetwork Topology Models

ILP Integer Linear Program

IoT Internet of Things

IP Internet Protocol

ITDK Internet Topology Data Kit

JDK Java Development Kit

JSON JavaScript Object Notation

LXC Linux Container

OSI Open Systems Interconnection

RAM Random Access Memory

SDN Software Defined Networks

TCP Transmission Control Protocol

WAN Wide Area Network

1 Introduction

Starting in the early 2000s, Cloud Computing has become the most popular solution for hosting and outsourcing of computation and storage. New payment models, dynamic resource allocations and ready to use services made it preferable to classic on site hosting in a data center.

While the sales of desktop computers and laptops are on the decline, smart devices are on the rise [Sta16]. Not only does this include smart phones, but other devices that can connect to a network (i.e. televisions, fridges, cars, etc.). According to multiple estimations, the market will consist of more than 30 billion IoT devices in the next 10 years [Nor16]. Although IoT devices are often sold as smart devices, they usually have limited computing and storage capacity. Therefore, such devices heavily depend on external computation, such as Cloud Computing.

Even though the cloud is a smart and valid tool for many problems, it does come with drawbacks. Since the cloud is located in data centers distributed around the globe, the communication latency to a server depends on the position and the characteristics of the connection used. This complication also applies to bandwidth, since bandwidth on those connections might be limited and are shared across all sent packages. Especially real time applications suffer from the high latency, since they can not guarantee a successful connection within their respective time bounds. Such applications do not benefit from the cloud and therefore seek for alternatives.

Fog Computing [BMZA12], a term coined by Cisco, is an alternative to the cloud. They proposed to move the computation and intelligence from the cloud to the network. Instead of a single data center located somewhere, the computation should be carried out on various devices within the network. Since the network starts from the device used and goes all the way to the cloud, the latency to devices in the network offers multiple options. Due to proximity, so called *fog nodes* reduce the latency and the bandwidth utilization.

Based on this proposal by Cisco other companies and research institutions founded the *OpenFog Consortium*[1], which aims to establish and standardize Fog Computing.

[1] OpenFog Consortium: https://www.openfogconsortium.org/

Therefore they published a Fog Computing reference architecture paper [Ope17] as a guide for future Fog Computing solutions. Even though this architecture is commonly referred to, there is no consistent standard yet. Still though Fog Computing or sometimes mixed with Edge Computing is a current trend in the area of distributed computing and hence the number of conferences and publications including Fog Computing rises.

Despite the limited availability of Fog Computing, the research is ongoing. In the last years more and more publications addressed Fog Computing or related fields such as Edge or Mist Computing, which also place computation closer to the user. Many of these publications evaluate their proposed work to show the benefit [AGC17; AH14; SCM15; WCU+15; XMR16]. Unfortunately, there are barely any testing tools available for Fog Computing. In most of these articles, the work is evaluated on fixed device topologies or simulate the application's behavior. This work attempts to solve this gap by proposing a new approach to test applications in arbitrary topologies by placing fog nodes and applications automatically to get a deployment cost optimal solution. Therefore this work introduces *EmuFog*. This tool can read in network topologies and identify the edge users can connect to. In such a network EmuFog places fog nodes containing the application software in a multilevel hierarchy. The final experiment including user devices, placed fog nodes, switches and links can be exported to network emulation tools to carry out the actual experiment.

This work is structured as follows. First, Section 2 provides the necessary background for future understanding. This includes Fog Computing, related work, the topology of the Internet and testing methods. Second, Section 3 defines the objectives of this work and the criteria to follow. Section 4 proposes a concept including a system model, workflow and algorithms to achieve the defined objectives. To carry out an evaluation of the concept, it has been implemented. This implementation is presented in Section 5. Section 6 provides an evaluation of the implementation. Finally, Section 7 concludes the entire work presented and proposes future work to further improve the concept.

2 Background

This section introduces the theoretical foundations of this work, including a definition and introduction to Fog Computing in Section 2.1 as well as related work in the area of evaluating Fog Computing in Section 2.2. Since Fog Computing is using the network of the Internet Section 2.3 provides insights in the topology of the Internet, including characteristics and models as well as real world topology databases and network generator tools. Finally, Section 2.4 introduces two tools that emulate software and network links in an artificial network for testing purposes.

2.1 Fog Computing

Over the past couple years Cloud Computing has been established as the preferred option to outsource computation power and storage capacity. Thereby all computation is moved to a dedicated data center of the vendor of choice. While this is often suitable and preferable, there are use cases where Cloud Computing may be infeasible. One common problem with Cloud Computing is the latency. Since it is deployed at predetermined geographical locations, the connection to the cloud depends on the one's location. In most cases, this leads to a latency too high for instance for real time applications. Another problem is the limited and shared bandwidth available; these drawbacks allow for innovating new approaches to tackling the issues.

Bonomi et al. [BMZA12] proposed a new approach called *Fog Computing*. Fog Computing tries to bring the computation and intelligence to the network itself, unlike the Cloud Computing paradigm of having a central data center. Devices, such as routers, switches, servers, and mobile stations, that are part of the network offer their resources to enable to computation capabilities within the network. They are referred to as *fog nodes*. Due to the proximity of the local network, Fog Computing can provide lower latencies than Cloud Computing can do.

A simplified overview of Fog Computing is depicted in Fig. 2.1. This figure illustrates the hierarchy of the Internet, including the well known cloud. At the bottom and hence the lowest level of the hierarchy are device as laptops, sensors, mobile phones etc. that can connect to some kind of network. This level is referred to as the edge of the network,

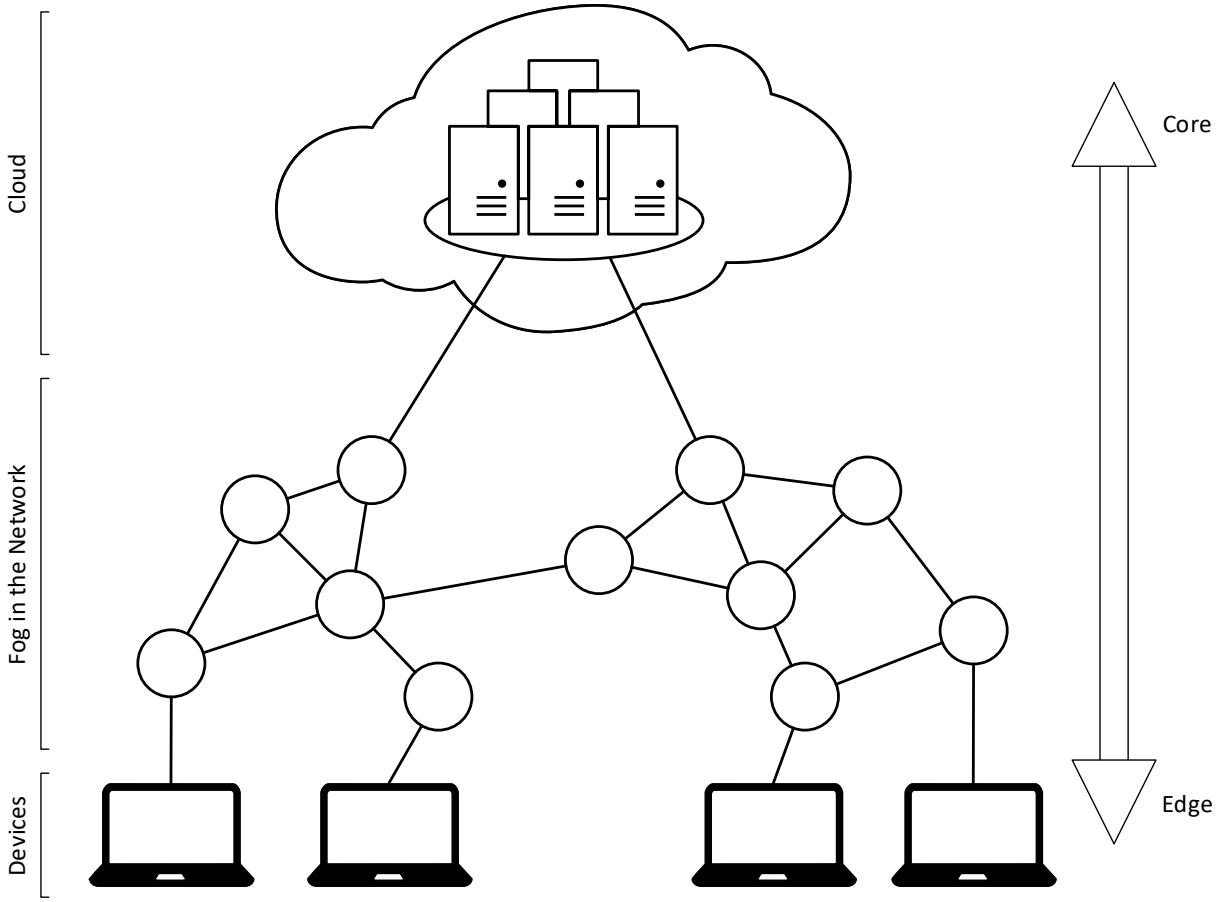


Figure 2.1: Overview of the Fog Computing Model

since these devices are usually connected to a single network and operated by end users. The top level of the Fog Computing model encompasses the cloud; the cloud is connected to different networks and not directly to devices. To get from the edge to the cloud it takes a path of network device connected to each other, establishing an end to end connection. All those network devices in between are possible fog nodes building the fog.

Various applications, such as stream processing, augmented/virtual reality, traffic control, smart homes/cities, etc., may benefit from Fog Computing [BMNZ14; YLL15]. Besides proposals there are also implementations that prove the benefit of Fog Computing by speeding up web site requests [ZCP+13] or improving face recognition [YLL15].

2.2 Related Work

The benefits of Fog Computing have been evaluated with different techniques in various papers. Those papers include evaluation of protocols such as MQTT [XMR16], CoAP or SNMP [SG16], but also test multi user applications, such as video streaming [AGC17] or Smart Gateways [AH14]. One downside to these evaluations is the testing scenario, because it is a manually crafted topology of either real devices or a building of network simulation tools. This may often lead to unrealistic scenarios or scenarios tailored to provide the optimum achievable in the respective work. More realistic scenarios can help to find bottlenecks, drawbacks or limitations of any kind people might not be aware of. Such scenarios can be built by testing tools specialized in the field of Fog Computing.

Gupta et al. [GVGB17] propose *iFogSim*, a toolkit to simulate and evaluate fog environments on a single device. Users can create their own topology and specify software modules to run in the network. Those modules get simulated in either the cloud or the network via Fog Computing based on the configuration, where the cloud is a central point in the topology. It produces different measurements on power consumption, network latency, congestion, and costs. The drawback of testing Fog Computing is the manual creation of a topology and the simulation approach with regard to not running the actual code of an application.

Besides iFogSim, there are other simulation tools for Fog Computing and IoT, such as the DPWSim [HLC+14], EdgeCloudSim [SOE17], and the commercial SimpleIoTSimulator[1]. Despite improving techniques for simulating application behavior, the results are only indicators and can not be equalized with executing actual software.

2.3 Topology of the Internet

On the highest level of abstraction the Internet consists of a myriad of different *Autonomous Systems* (AS). An AS is a network managed autonomously by its provider. The connection of two autonomous systems uses the *Border Gateway Protocol* (BGP) on top of the well known TCP. As the connection of autonomous systems requires a contract between the respective providers, some paths are routed though multiple autonomous systems as a transit network.

Currently, there are no complete screenshot of the Internet's topology available. This is not just due to the fact that there is always change in the network but also due to secrecy.

[1] SimpleIoTSimulator: https://www.smplsft.com/SimpleIoTSimulator.html

Since the network connection on the AS level is not managed by a single organization but instead is based on individual connections the topology can only be tried to measure. There are several projects trying to retrieve the AS level topology by analyzing the BGP routing. Each of the autonomous systems is considered a router level topology, since it contains the routers user can connect to. As the router network is a business asset of the respective provider they usually do not provide them to the public.

As mentioned before, there are projects that are trying to provide Internet topologies on the AS and router level through the analysis of routing tables, BGP, and published topologies. Besides the two popular projects introduced below, there are many more providing similar information.

- The *Center for Applied Internet Data Analysis*[2] (Caida) does various kind of research in the networking field including the Internet topology. This includes publicly available measured AS, IPv4 and IPv6 router level topologies.
- *Topology Zoo*[3] [KNF+11], published through the University of Adelaide, provides AS level topology datasets and visualization.

2.3.1 Modeling Approaches

In order to describe the topology of the Internet or a network in general, researchers have come up with different models having certain properties to formulate the topology. Over the years, multiple models with various properties have been presented. Overall, the models are different, but they can be grouped into three main categories: *random-*, *structural-* and *degree*-based.

First, the most basic approach is the idea of random graphs. These graphs have a random placement of the nodes or maybe use some kind of clustering of them. Another random factor might be the edges of the graph. The Waxman model [Wax88] is one of the most popular models within this category. In this model, nodes will either be randomly distributed on a plane or placed closely together using a grid layout and assigning a different number of nodes per grid-cell. The connection of two nodes is a probability based on the distance between the two of them.

The structural-based model was designed by scientists due to their dissatisfaction of the outcome of the random graphs. Moreover, this model was also developed based on their interest of the structure of actual networks and the Internet. *Transit-Stub* [CDZ97] and *Tiers* [Doa96] are two popular models of this category. Both of these models favor

[2]Center for Applied Internet Data Analysis: https://www.caida.org/
[3]Topology Zoo: http://topology-zoo.org/

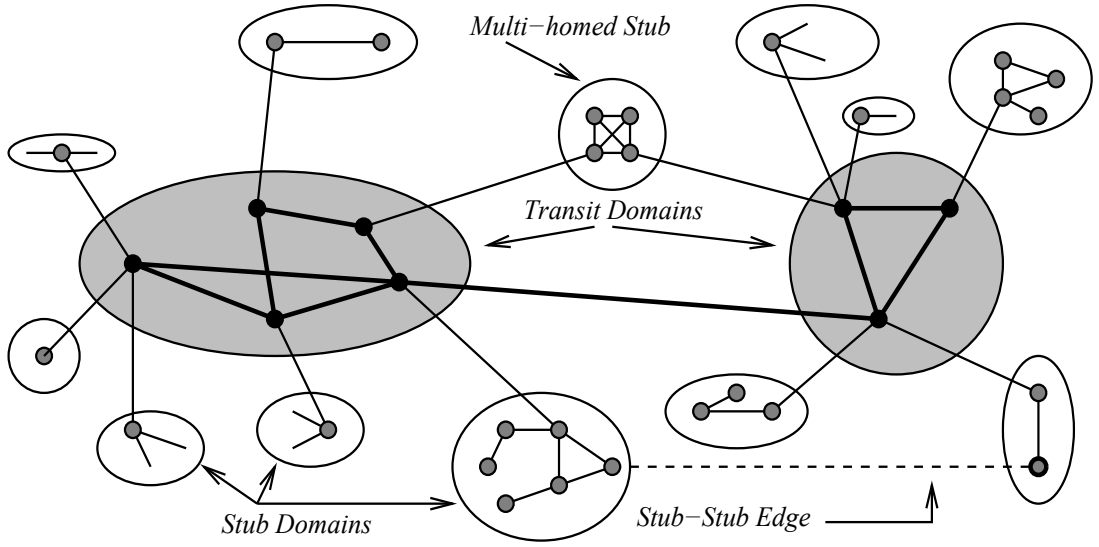


Figure 2.2: Hierarchical Structure of the Transit-Stub Model [CDZ97]

the hierarchy of the Internet as a foundation of their placement. Figure 2.2 depicts an example of the structural-based model, Transit-Stub.

Third, since they seem to model the Internet the closest, most recent research publications use a degree-based model. Faloutsos et al. [FFF99] discovered that the topology of the Internet obeys power-laws in the form of $y \propto x^{\alpha}$ e.g. for the distribution of in and out degree. They based their finding on three snapshots of the Internet topology in the late 1990s. Even though the Internet has grown and changed over time, the power-law is still the latest and most commonly used attempt to model the network. Medina et al. [MMB00] were able to prove that the common random- and structural-based models, such as the Waxman and Transit-Stub do not match the properties of the power-laws.

2.3.2 Topology Generators

Over the years, several topology generators have been proposed to aid the various usages of network topologies, such as, protocol testing, simulation application behavior, and performance measures. This section will present a subset of all available generators and thereby focus on relevant generators that are commonly used in research.

BRITE

The *Boston Representative Internet Topology Generator*[4] (BRITE) [MLMB01] has been one of the most popular network topology generators in recent years. Medina et al. used the generator for their evaluation on power-laws [MMB00]. Thus, the generator supports different models to generate from. It supports the Waxman model and three degree-based models on an AS and router level. Barabási and Albert [BA99] proposed a model satisfying the power-laws by Faloutsos et al. and also an improvement [AB00] of their own model. Besides the two models by Barabási and Albert, there is the *Generalized Linear Preference* (GLP) model proposed by Bu and Towsley [BT02] which also satisfies the power-laws.

IGen

Unlike most of the other generators mentioned in this section, the *IGen Generator*[5] [QSFB09] only produces router level topologies. The authors claim that even degree-based models cannot model the Internet accurately, since they do not take design heuristics into account. The actual network of the Internet is built by people using certain practices to optimize the overall functionality. Based on this assumption, Quoitin et al. use heuristics to minimize the latency, optimize bandwidth, etc. to construct a graph. But, there is a lack of evidence that IGen produces more accurate graphs than other generators.

Inet

Another popular topology generation is the *Inet-3.0*[6] [WJ02]; in the most recent version, which only generates AS level topologies. Similar to other topology generators, Inet uses a degree-based model based on the idea of power-laws in the Internet. Winick and Jamin provide an evaluation of Inet-3.0, the predecessor Inet-2.0, and snapshots of the Internet. Graphs generated by Inet satisfy the power-laws by Faloutsos et al. and also match the Internet reasonable in other metrics provided as hop count, path length, and clustering coefficient.

[4] BRITE is available at https://www.cs.bu.edu/brite/
[5] IGen is available at http://igen.sourceforge.net/
[6] Inet-3.0 is available at http://topology.eecs.umich.edu/inet/

GT-ITM

In comparison to the generators mentioned before *Georgia Tech Internetwork Topology Models*[7], the GT-ITM is a generator using the Transit-Stub model [CDZ97], hence uses a structural-model. In Figure 2.2, you can see two types of domains, the Transit- and the Stub-Domains. Stub-Domains should only handle traffic that is sent or received by one of its nodes. The Transit-Domains, on the other hand, do not generate traffic as there are no clients connected and can be seen as network to forward traffic to the respective Stub-Domain. Both domains are generated using the Waxman model and the connection between such is based on parameters provided by the user.

Tiers

The *Tiers Generator*[8] [Doa96] shares a lot of properties with the Transit-Stub model. Both are structural models with several levels or tiers. In comparison to the Transit-Stub, Tiers has only one top level network, the *Wide Area Network* (WAN). To generate the topology, Tiers uses a minimal spanning tree which leads obviously to one root node and a strong hierarchy. The main drawback of this approach seems the unlikeliness of a tree topology in the Internet.

2.3.3 Comparison of Topology Generators

Comparing graphs produced by network topology generators to the actual Internet is a difficult task. There are two important challenges: metric usage and the topology of the Internet. It is not obvious what metric is the right to compare. Depending on the model the generator uses, different metrics might be applicable. Second, a topology of the Internet to compare to. AS level topology snapshots can be measured by the BGP which connects different AS. Router level topologies on the other hand are hard to get as they are part of the business decisions of Internet providers.

This section will identify a suitable topology generator for the fog emulation framework based on evaluations. It should model the Internet topology closely on AS and router level. Section 2.3.1 already introduced the degree-based category and how it emerged from previous attempts to model the Internet. Tangmunarunkit et al. [TGJ+02] have shown in an evaluation that degree-based topology generators not only match the power-laws better than structural-based models, but they also model the large scale of

[7] GT-ITM is available at https://www.cc.gatech.edu/projects/gtitm/
[8] Tiers is available at https://www.isi.edu/nsnam/ns/ns-topogen.html

the Internet better, measured by eight different metrics. Even though it is unclear what the best metric or category is, most publications focus on degree-based models.

Bu and Towsley [BT02] evaluate their GLP model as well as the both models by Albert and Barabási all included in BRITE and furthermore the Inet generator. For the evaluation, they use an AS level topology gained from Route Views[9]. After Medina et al. [MMB00] showed that power-law generators produce more similar graphs to the Internet than random- and structural-based generators, Bu and Towsley use the power-laws as a metric as well as the characteristic path length and the clustering coefficient defined by Watts and Strogatz [WS98]. The improved model by Albert and Barabási shows a high similarity with the real world topology.

Heckmann et al. [HPSS03] use a different approach to compare generators not relying on the power-laws as a metric, but instead use a set of connectivity properties. For their evaluation, they use two router-level topologies from *AT&T* and the *German Research Network* (DFN) instead of AS-level snapshots. Both topologies got evaluated using the GLP, Tiers and GT-ITM model. Tiers performs the best in this evaluation with a score of 0.998 and 0.995 on a scale from 0 to 1 for the DFN and AT&T topology respectively. BRITE scores a high similarity of 0.972 rsp. 0.951 and GT-ITM of 0.966 rsp. 0.879.

2.4 Network Emulation

Testing applications and protocols in a distributed environment can be problematic, since they do not solely rely on the developer's local machine. Other devices, servers, and various connections with different properties have an influence on the result and behavior of such applications. This makes testing tools a necessity. So far there are three categories of testing approaches available: live testing, simulating, and emulating.

Live testing provides the best results as it is the closest to the final real world deployment, but it is also the most expensive and sophisticated. In the area of distributed computing, especially Fog Computing, there are live testing facilities available like *Iot-LAB* [PBG+13] with over 1,000 nodes across four cities in France or *SmartSantander* [SMG+14] with a variety of sensors etc. distributed across the city of Santander, Spain. Gluhak et al. [GKN+11] provide a survey of such IoT testing facilities for real word testing. Besides the costs, another drawback is the inflexibility of this approach. Changing the topology or devices used in such an experiment can be challenging.

[9] Route Views: http://www.routeviews.org/

In contrast to live testing, simulation tests can be executed fast and easily on every machine. But since simulation does not execute the actual source code of the protocol to test, the results may not be as accurate as live testing. Emulation involves filling the gap between live testing and simulation. It executes the source code, usually in an virtualized environment, instead of the actual hardware to deploy on in the end. Hence, emulation can achieve a good trade-off between the cost of an experiment and the accuracy of results to expect.

Lochin et al. [LPD11] present an extended comparison between the different approaches, their pros and cons as well as guidance on what to use depending on the test scenario. Also, they discuss the different emulation models such as: static, event-driven or trace-based, and provide a selection of available network emulation tools. Static models have to be configured before the emulation starts and stay exactly the same during the execution. In contrast to static models, event-driven models change their characteristics based on events such as clock ticks, packages, errors, etc. Trace-based models measure a real world network and obtain the characteristic. Those can be applied to the emulated network as in the actual network.

Since live testing is expensive and inflexible to test and simulation just does not provide accurate results, emulation is the best option to test Fog Computing applications and protocols. This section will cover two recent emulation tools, MaxiNet and CORE, using the virtualization approach in more detail.

2.4.1 MaxiNet

Wette et al. [WDS14] proposed *MaxiNet*[10] as an emulation framework for Linux to run *software defined networks* (SDN) across several physical machines. It enables a user to emulate large artificial networks on a private computer. Therefore, the switches and hosts can be placed in the topology and connected via links. This helps debugging protocols and applications efficiently instead of an actual deployment. Hosts can run commands and applications installed on the underlying machine including Docker container, which allows to the user to build a Docker container consisting of the desired test environment. MaxiNet is a distributed version of *Mininet* by Lantz et al. [LHM10] that provides the network emulation using a single physical machine.

Mininet is an emulation tool aiming to be especially lightweight by using standard Linux features. Each host in the network has its own namespace *Linux container* (LXC) to store its network and execution state. Such a LXC namespace is isolated from the execution and file system of the underlying Linux system and other namespaces making

[10] MaxiNet is available at https://maxinet.github.io/

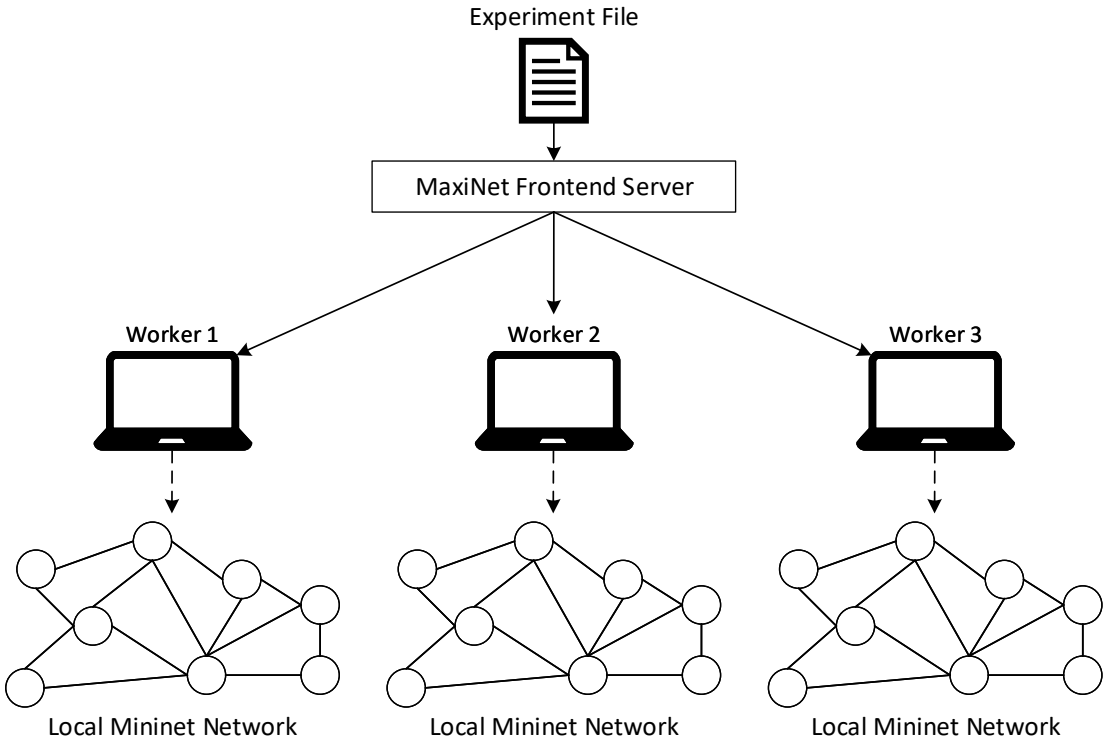


Figure 2.3: The MaxiNet Architecture for Distributed Emulation

it independent. This is crucial as it allows multiple hosts to run the same application of the underlying system without interference. Links between hosts are implemented with Linux bridges and can be me modified in terms of limiting bandwidth, applying latency, etc. by the traffic control `tc` library. Experiments can be written in python files including the topology of the network as well as commands to run after the initial setup of the network.

The main drawback of Mininet is the limitation to one physical machine; there is a limit in emulated hosts depending on computing capabilities of the machine and the commands to run. MaxiNet tries to resolve this issue by distributing a network topology over several distributed physical machines, referred to as workers. In their work, they present an evaluation of MaxiNet showing that they can run to 3,200 hosts on 12 physical machines. An overview of the architecture is illustrated in Fig. 2.3. The MaxiNet Frontend Server manages the connection with the different workers running MaxiNet to keep them in a pool of idle workers. By starting an experiment file on the Frontend Server, the server distributes the emulated hosts to the different workers by partitioning the undirected graph using METIS [KK95]. Each worker is running an instance of Mininet emulating everything associated to that worker. To route the

inter-topology, traffic workers communicate directly via *Generic Routing Encapsulation* (GRE) tunnels.

2.4.2 Common Open Research Emulator

Another emulation framework is the *Common Open Research Emulator*[11] (CORE) by Ahrenholz et al. [ADHK08; Ahr10] available for Linux and FreeBSD. CORE itself is an improved modification of the IMUNES [ZM04] emulator by the University of Zagreb. Like MaxiNet, CORE aims to emulate artificial networks as lightweight and scalable as possible on one or multiple devices. Therefore, CORE uses a hybrid solution of emulation and simulation. Network connections are simulated and the hosts are emulated to actually execute commands and applications. Hence, CORE focuses on the OSI layer 3 and up to test applications and protocols. Nevertheless it can be combined with MANET, a network emulator for the networking layers including all kinds of network congestion, to emulate the network too [AGA11].

CORE's host virtualization is using jails for FreeBSD to isolate the host environment from the operating system and the previously mentioned LXC namespaces in Linux. Both ensure to have their own running processes, network stack, and file system. Ahrenholz et al. also provide an evaluation of the different technologies comparing FreeBSD jails, Linux namespace, and its historical predecessor OpenVZ. In this evaluation FreeBSD can achieve a better result, measured in packages per second over the two Linux alternatives.

Large topologies can be distributed around multiple physical devices to ensure enough computational power. The CORE GUI is running as a controller partitioning the topology and distributing it to CORE Daemons. Similar to MaxiNet, each Daemon hosts a subset of the topology and routes traffic to other Daemons via direct tunnels established at deployment time.

[11] CORE is available at https://www.nrl.navy.mil/itd/ncs/products/core

3 Objectives

As discussed in Section 2.2, there is a variety of research work in the area of Fog Computing. Many of the researchers provide an evaluation of their work using different test scenarios. Some simulate the behavior, some emulate their software, and some even deploy their software in a real world test environment. However, most of these works suffer from the fact that they use a static topology often unlikely to model the real world, e.g. star topologies or fully connected networks.

This work aims for a new approach to test software in the area of Fog Computing. Instead of using handmade topologies where fog nodes have to be placed manually to achieve realistic results, this work should help to automate this process and thereby make it easier and faster to test. Topologies can still be created manually, but are also generated by additional tools thereby increasing the number of possible test scenarios. The user should be able to evaluate its own or existing software in a multi level fog environment. The software runs on configurable devices placed in the network connected to automatically placed fog nodes, thus providing flexibility for different use cases.

The following criteria should be considered in the new testbed:

- *Scalability:* Despite the fact that most evaluations use small networks for their test scenario, this approach should be able to scale with the size of the network. Due to the growth of networks, focusing on smaller networks would be a critical limitation. Hence, it should be able to handle networks in the size of several thousand nodes.
- *Extensibility:* As discussed in Section 2.3, there are many ways to get an either artificial or real world topology on the Internet. It should be possible to extend the testbed to use different topologies and formats as an input. The output should also be easily expendable to generate different formats for the variety of different emulation tools.
- *Flexibility:* Since every evaluation uses a different scenario, different hardware, and different software to emulate, the users should be able to tune the testbed to their needs. This includes different fog node types to use, executable software, tuneable virtual hardware configurations, etc.

- *Platform-independent:* As the approach should be expendable for new network topology and emulation formats, the entire testbed software should run independently of the operating system. This way it can provide a workflow from start to finish in a single environment.

4 Concept

In this section a new approach for a Fog Computing testbed will be presented. Its objectives and criteria have been defined in the previous section. Fig. 4.1 shows a simplified overview of the workflow. First, the workflow gets briefly explained and second, each step is covered in more detail in its own section.

Beginning at the first and most left point there are network topologies. They can be either real world datasets or artificial graphs as described in Section 2.3. Those topologies get read in by the testbed and get converted into a graph structure that is independent of the input format modeling the necessary information. All following operations apply to this graph structure leaving the input data unchanged.

Once the graph is read in the graph structure, the edge of the network has to be identified. The graph structure uses a simple model dividing the graph into two parts: the edge of the network and its backbone. To identify the edge, it uses a classification based on the routing of packages and the degree of the vertices. Having a separation of edge and backbone in the network is crucial for the upcoming steps, as they rely on this information.

After defining the edge of the network, devices can be placed in the topology and connected to the edge. Devices such as laptops, mobile phones, and sensors get distributed across the network and connected to one edge node each. The connections starting at these devices need to be handled by the fog nodes placed in the next step.

With the devices placed in the network, there is a need to place fog nodes that they can connect to. Such fog nodes should be placed cost optimally within a given distance, e.g.

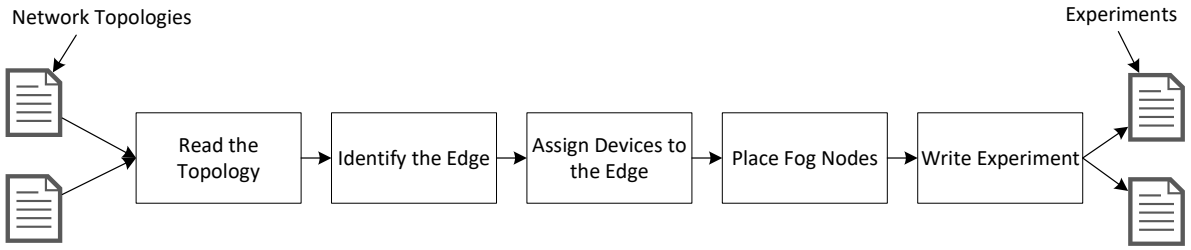


Figure 4.1: Overview of the Workflow

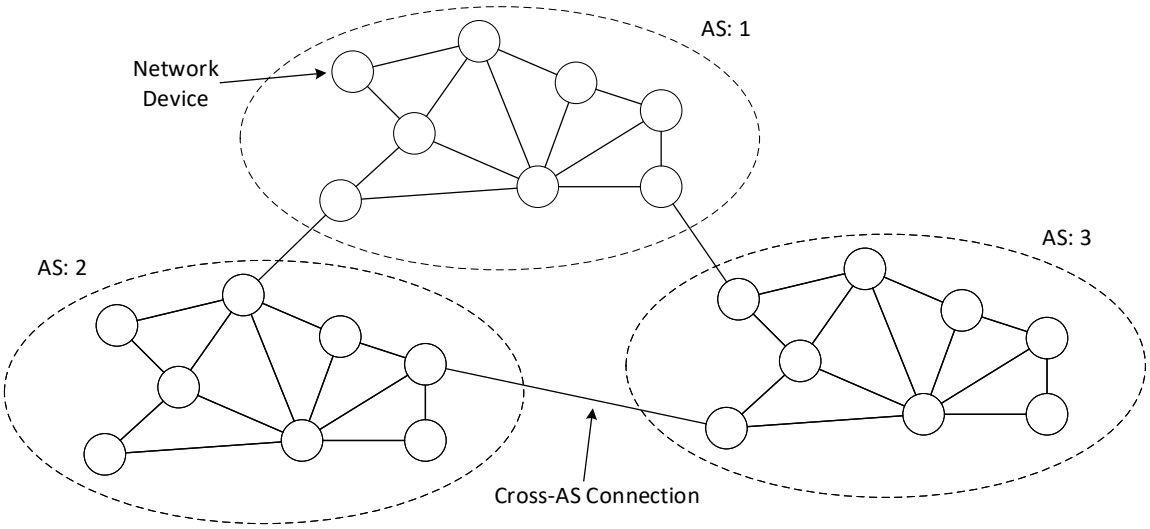


Figure 4.2: The Network Model used in the Proposed Fog Emulator

latency from the devices. Since placing them optimally is a challenging and complex problem, this approach uses a heuristic to provide a good solution within a reasonable time.

Finally, after placing the fog nodes in the graph structure, the output files can be generated. The idea is to write files able to start an experiment using an emulation tool like the ones discussed in Section 2.4. Similar to the topology reader, the writer should be extensible for further data formats.

4.1 Network Model

To ease the understanding of the following detailed descriptions of each step, knowledge of the underlying network model is helpful. A simple demonstration of the network model is depicted in Fig. 4.2. The overall network is separated into multiple autonomous systems. On the highest abstraction level, the graph consists of vertices representing autonomous systems and edges representing cross-AS connections. In this example, there are three autonomous systems drawn surrounded by a dotted line with their respective unique identifiers.

Each of the autonomous systems is its own network where vertices are actual physical devices that are connected to another. There is no cut set between two autonomous systems because each device is associated to exactly one autonomous system. An autonomous system consists of three subsets of vertices or nodes. First, the devices

connected to the network. This set contains devices (i.e. laptops, mobile phones, etc.) that can connect to a network. But it does not contain routers, switches, and other network technology devices. Second, the access network. All nodes within the access network allow end users to connect to the entire network. Those nodes are called access points. Since they are usually at the edge of the network, they also get referred to as edge nodes. The terms access points and edge nodes are identical and will be used interchangeably throughout this work. Third, the backbone network. Nodes being part of the backbone do not connect to end users directly as edge nodes do. They route traffic generated somewhere in the network to its destination. Therefore they handle in comparison to edge nodes more traffic.

4.2 Read the Network Topology

The first step is to read in the input data. In this case the input is some kind of a network topology. Any undirected graph can be used as such a network topology input. Even though there are standardized graph formats like GraphML [BEH+02], GML [Him97], and GXL [WKR02], most generators or network databases use their own format which makes it impossible to write one reader fitting all needs. Instead of a universal reader supporting all data formats, it should be possible to easily extend the existing reader by the required format. To build a network topology from the reader, the graph structure has to offer an easy to use interface to create nodes and connections.

4.3 Identify the Edge of the Network

The overall algorithm to identify the edge of the network consists of three steps. Starting on a graph $G = (V, E)$, the algorithm expects to have edge nodes only. In case there already exist backbone nodes at this time, they will be included and connected to newly discovered in the last step. Every step of the algorithm will increase the backbone and as a result shrink the edge. In the following sections, the sub set $B \subseteq V$ will depict the nodes of the backbone which will be carried through the different steps of the algorithm. The sub set $A \subseteq V$ is the set of access points/edge nodes that has no cut set with the set of backbone nodes $A \cap B = \{\}$. Algorithm 4.1 shows the necessary three steps of the overall algorithm.

Algorithm 4.1 Identify the Edge of the Network

```
1: procedure IDENTIFYEDGE(G = (V, E))
2:     B ← SELECTCROSSASCONNECTIONS(G)
3:     B ← SEEKHIGHDEGREENODES(B, G)
4:     B ← CONNECTASBACKBONE(B, G)
5:     return B
6: end procedure
```

4.3.1 Select Cross-AS Connections

Connections in the graph that connect two different autonomous systems have to handle a lot of traffic in the network, since they are the gateway for a variety of connections from one AS to another. Traffic routed between different systems uses the Border Gateway Protocol on such connections. In contrast to the internal routing within an AS, there are just a few cross-AS connections which makes them the bottlenecks that often throttle the overall connectivity between autonomous systems. Both ends of this connections are highly unlikely an edge node users can connect to. More than likely, those endpoints are high performance switches to handle the load. Since a cross-AS connection might be the only connection between the two of them, traffic routing has to route the packages via this connection. Therefore the algorithm adds all endpoints of cross-AS as switches of the network's backbone.

The first step is depicted in pseudo-code in Algorithm 4.2. Input parameter is the unmodified graph $G = (V, E)$. Starting from an empty set of backbone nodes B (line 2), the algorithm iterates over all edges in the set of edges E and checks whether the source's s and destination's d AS are unequal (line 3–4). In case they belong to different autonomous systems, source s and destination d will be added to the set of backbone nodes B (line 5).

Algorithm 4.2 Selecting Cross-AS Connections

```
1: procedure SELECTCROSSASCONNECTIONS(G = (V, E))
2:     B ← {}
3:     for all e = (s, d) ∈ E do
4:         if s.AS ≠ d.AS then
5:             B ← B ∪ {s, d}
6:         end if
7:     end for
8:     return B
9: end procedure
```

4.3.2 Seeking High-Degree Nodes

In the first step, the algorithm added endpoints of cross-AS edges to the backbone of the network which are are crucial for cross-AS routing. Besides cross-AS endpoints, there are other nodes in the topology that have to handle an above-average amount of routing. A node with a high degree in the graph has to manage many connections which is likely to be a backbone node just routing packages, whereas an edge node will have a rather small degree just connecting to the backbone of the AS.

Algorithm 4.3 sketches the second step of the edge identification algorithm. Starting with the input from the first step, the algorithm first calculates the average degree of nodes in the set of V (line 2). Based on the average node degree avg, all nodes that are not yet part of the backbone get compared to the average. If the degree of a node is higher than the average, it gets added to the set of backbone nodes B (line 3–7).

Algorithm 4.3 Add High Degree Nodes to the Backbone

```
1: procedure SeekHighDegreeNodes(B, G = (V, E))
2:     avg ← CalculateAverageDegree(V)
3:     for all v ∈ V \ B do
4:         if v.Degree ≥ avg then
5:             B ← B ∪ {v}
6:         end if
7:     end for
8:     return B
9: end procedure
```

4.3.3 Connect the Backbone of an AS

With the first two steps, the algorithm creates a subset $B \subseteq V$ of nodes that are part of the backbone. The result of the first step can be, and most likely will be, a set of backbone nodes that are not connected with each other as exemplary depicted in Fig. 4.3. It can be seen that the detected backbone nodes, colored in gray, are not connected to another via backbone nodes only. In this example, there is at least one edge node between the two partitions. Within an AS, there is only one backbone that connects the different edge routers of the AS. Having a partition in the backbone of the AS will result in a connection between the two partitions consisting of an edge node. Considering all the traffic that is routed in the backbone, it is unlikely that this traffic will flow through the edge of the network. In order to establish such a backbone, all nodes of B have to be connected.

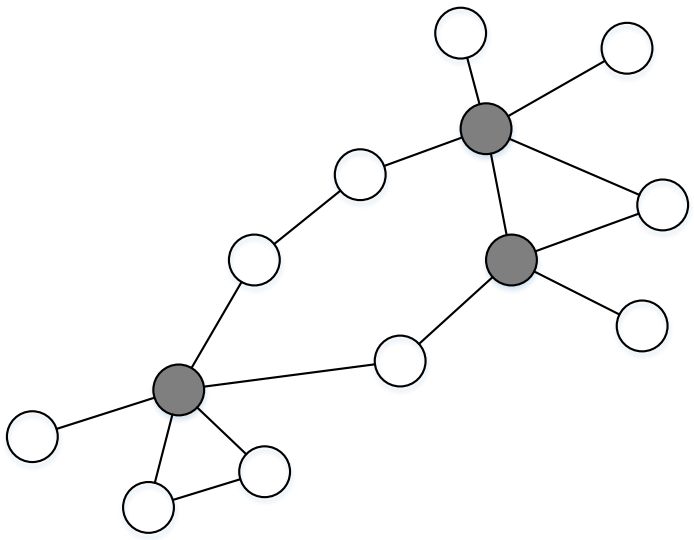


Figure 4.3: Partitioned Backbone After First two Steps

The third and last step of the algorithm is listed in Algorithm 4.4. The overall idea is based on the Breadth First algorithm (BFS), which iterates through the graph starting at a certain point and adding the direct neighborhood to a queue. This way the algorithm continues until the queue is empty and all nodes have been processed. From the previous two steps the algorithm already gets a set of backbone nodes B.

The basic idea of the algorithm is visualized in Fig. 4.4. Starting from the initial situation in Fig. 4.3, the algorithm picks a random backbone node from the set B. In this example, the algorithm selects the bottom left backbone node as can be seen in (a). Each neighbor node in the direct neighborhood of the currently processed node is added to the queue of nodes to process and the predecessor gets set to the current node. The direction of the predecessors is visualized by arrows pointing to the predecessor. In (b) all nodes up to the second neighborhood of the starting node have been processed. The algorithm found a backbone node whose predecessor is an edge node. Since we started from a backbone node it was reached by a path containing edge nodes and therefore a partition might exist. By following the trace of predecessors back to the next backbone node, the algorithm can connect the two partitions by adding all edge nodes on this path to the set of backbone nodes B. The result is depicted in (c). Finally, the algorithm continues as in (d) and sets the predecessors as shown here for the third neighborhood of the starting node.

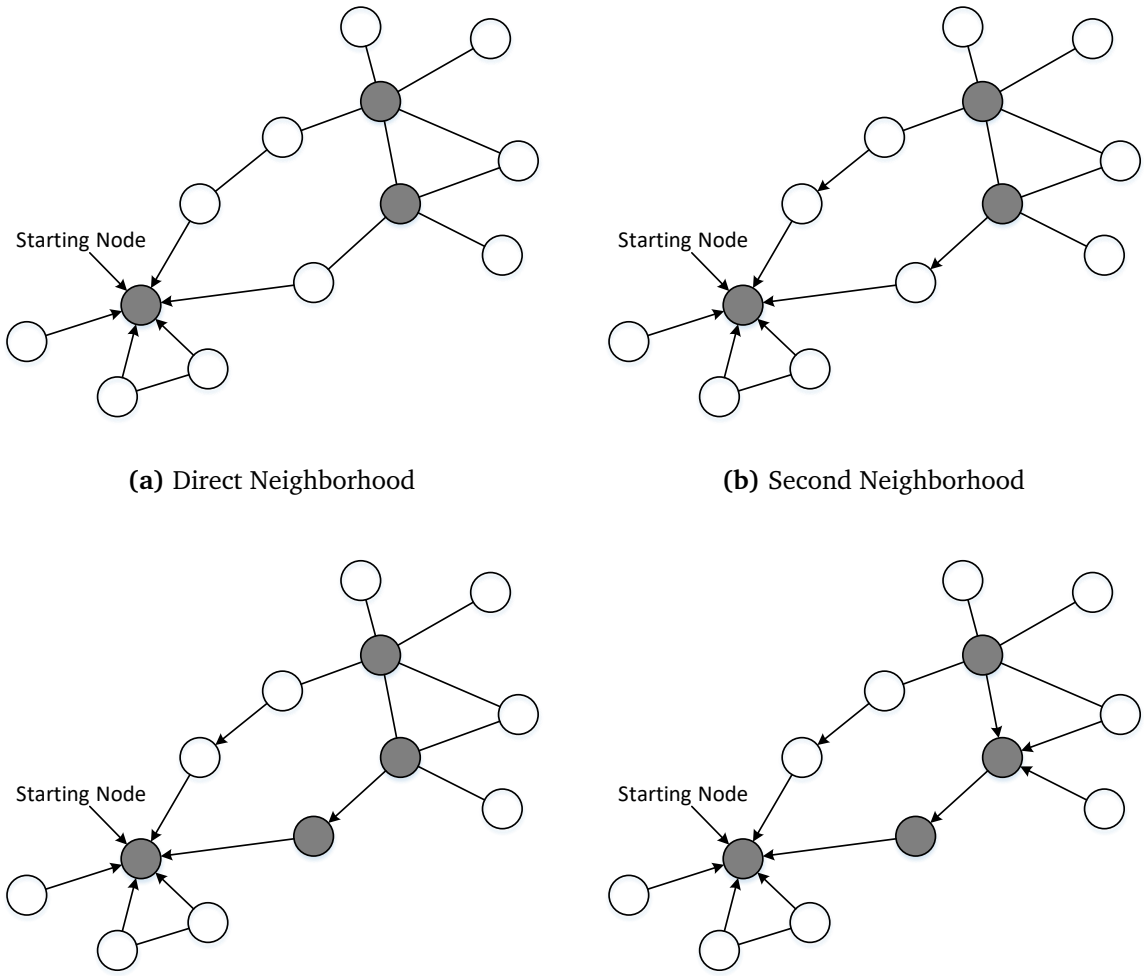
Since the Breadth First algorithm iterates through the entire graph, there is no limitation in which node to start with. The proposed modified version of BFS starts with a random

Algorithm 4.4 Connect the Backbone of an AS

```
1:  procedure ConnectASBackbone(B, G = (V, E))
2:      b ← b ∈ B                                   // pick any random node in B
3:      Q ← {b}
4:      while Q ≠ {} do
5:          c ← Q.Dequeue()
6:          for all n ∈ N_1(c) do
7:              if n ∈ Q then
8:                  if c ∈ B ∧ n.Predecessor ∈ V \ B then
9:                      n.Predecessor ← c
10:                 end if
11:             else
12:                 n.Predecessor ← c
13:                 Q.Enqueue(n)
14:             end if
15:         end for
16:         if c ∈ B ∧ c.Predecessor ∈ V \ B then
17:             p ← c.Predecessor
18:             while p ∈ V \ B do
19:                 B ← B ∪ {p}
20:                 p ← p.Predecessor
21:             end while
22:         end if
23:     end while
24:     return B
25: end procedure
```

node of the backbone set B (line 2). All nodes yet to be processed are stored in the waiting queue Q. For each node in the queue, the algorithm does two things. First, iterate the neighborhood to add new nodes to the queue as the Breadth-First algorithm does (line 6–15) and second, connect disjoint partitions of the backbone (line 16–22).

As mentioned before, the algorithm starts with an arbitrary backbone node of B. From this node, the algorithm expands through the network by processing each node. A queue Q is used to store the nodes still to be processed and their order, and is initialized with the starting node (line 3). As long as there are still nodes to process in the queue, the algorithm removes the first node from the queue and iterates the direct neighborhood of this node $N_1(c)$ (line 6–15). In case a neighbor node is not yet it the queue, the node gets added to the queue and the predecessor gets set to the currently processed node c (line 11–14). If the neighbor node is already in the queue and its predecessor



(a) Direct Neighborhood

(b) Second Neighborhood

(c) Connect Backbone by Following Trace

(d) Third Neighborhood

Figure 4.4: Exemplary Connection of Backbone Partitions

points to an edge node and additionally the currently processed node is a backbone node, the predecessor gets updated to point to the current node c (line 8–10). That way unnecessary paths via edge nodes can be reduced. This can be seen in Fig. 4.4 (d) where the predecessor of the backbone node points to another backbone node instead of an edge node.

To connect two partitions, the predecessor field of each node is used. This is only necessary if the currently processed node c is part of the backbone B and its predecessor is not (line 16). In this case the current node was discovered on a path of edge nodes, but since we started from the backbone there has to be a backbone on this path following

back to the starting node. By following the path back using the predecessor field until there is the backbone again, two partitions get connected (line 17–21).

When the queue is empty it is guaranteed that no node is left in the graph that has not been processed yet. As a result there is only one backbone of this autonomous system left.

4.4 Placing Devices in the Network

After the access point and the backbone got identified in the previous section, client devices that connect to fog applications can finally be placed at the edge of the network. A device represents basically every device like sensors, mobile phones or laptops that are connected to an access point. Each device is connected to exactly one access point and to no other node of the graph. Such device can be placed manually in the graph by the interface of the graph structure, but also can be generated as shown in the implementation in Section 5. Access points without devices get ignored in the fog placement algorithm.

4.5 Fog Node Placement

In the previous sections, the graph got divided into three disjoint sets: the access points, the backbone of the network, and the devices connected to the access points. So far there is no Fog Computing in this network. As a next step fog nodes have to be placed in the topology in order to enable Fog Computing. Such fog nodes can only be placed within the network and not on client devices. Therefore, only access points and the backbone excluding the devices of the access points are considered.

This section structures as follows. The problem of fog node placement will be introduced in Section 4.5.1 and to approximate this problem Section 4.5.2 proposes a heuristic algorithm.

4.5.1 Problem

The placement of fog nodes is a challenging task due to the fact that there is no best practice on where to place them in the network topology. Depending on the scenario optimal placement might vary. So far there is no guidance or proven research in this area. Therefore additional information by the user is required. The user has to provide possible

fog types to place. Such a fog type consists of computation capabilities, connection capacities, executable software, and dependencies between fog types to build multi level hierarchies.

Starting with a graph $G = (V, E)$ and its sub set $A \subseteq V$ of all edge nodes the problem is to find a placement of fog nodes $F \subseteq V$ such that every edge node in A is connected to a fog node in F. This problem could be solved by placing a single fog node somewhere in the topology, but this does not reflect the idea of Fog Computing. To limit the distance from an access point to the next fog node there has to be a threshold. Such a threshold could be based on the latency, number of hops, or other measurements. Having such a threshold leads to a local range of an access point within which a fog node has to be placed. All nodes within that threshold are possible fog node placements for this access point.

Besides the problem where to place fog nodes in the topology, there is also the problem of choosing the appropriate fog node type. Fog nodes can be of different types based on their placement. In a real world scenario a fog node could be a home based router, a mobile radio station, a switch in the network etc. Hence, the type of a fog node has to be determined, too. Since there is no information about different protocols or the underlying hardware, the type identification is limited. The type of a fog node will be determined based on the maximum connections available and the deployment costs of the respective fog node type. The goal is to find a set of fog nodes covering all devices with minimal deployment costs for the fog nodes.

To illustrate this problem, a simple graph is depicted in Fig. 4.5. The two white vertices are access points and the remaining gray vertices are part of the backbone. Devices have been omitted for the sake of simplicity. In this example, there are two fog node types available. One with a capacity of 1 and costs of 1 and another one with capacity of 2 and costs of 1.5. Further should the threshold be the number of hops.

For the first example, the threshold is 1. The outcome is depicted in Fig. 4.6. Both access points have a dotted circle surrounding them to visualize the range of possible fog nodes. In this case, there is no overlap between the two ranges and hence the result is having two fog nodes of the first type directly at the access point. The overall deployment costs are 2.

Increasing the threshold from 1 to 2 results in a different graph depicted in Fig. 4.7. Now the two ranges intersect in a single vertex of the backbone. Since the goal is to place cost optimal the result is having a single fog node of the second type. Hereby the overall costs can be reduced from 2 to 1.5.

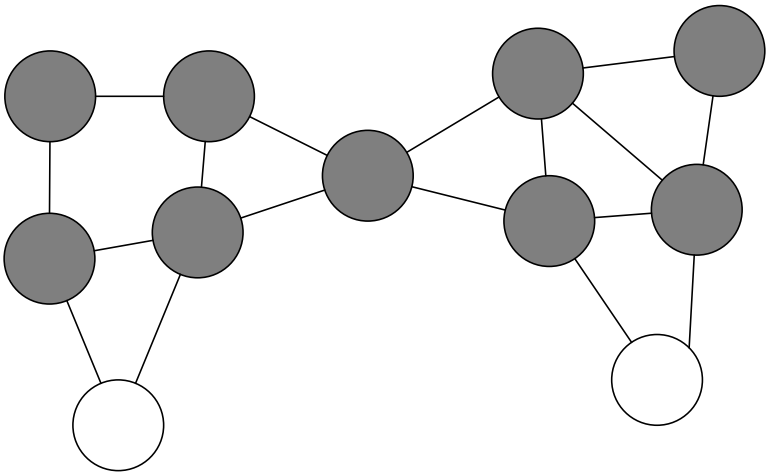


Figure 4.5: Extract from a Network Graph

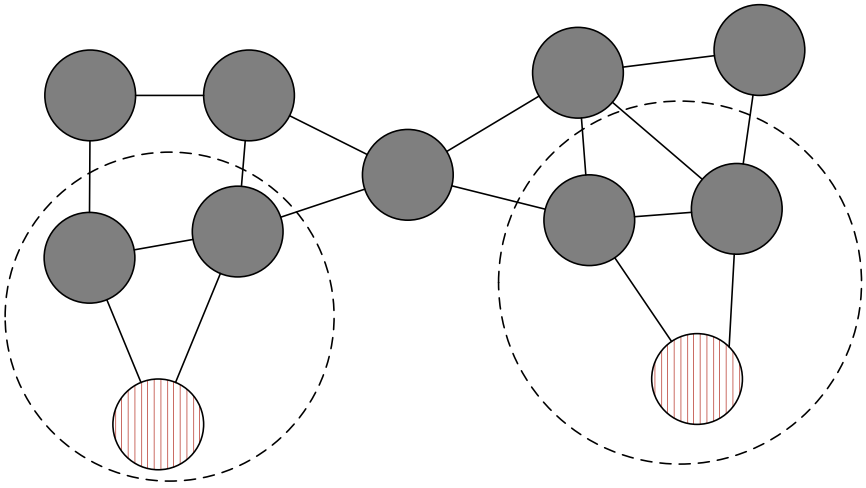


Figure 4.6: Exemplary Placement of Fog Nodes using the Threshold 1

4.5.2 Heuristic

The heuristic presented in this work is a greedy algorithm looking for possible fog node placements with lowest average deployment costs. Therefore, the algorithm calculates a list of edge nodes for all possible fog nodes within their range given by the threshold t. With that information, they can determine the optimal type of fog node to deploy based on the maximum connections and their respective deployment costs. Each iteration takes the cost optimal placement in the current state and adds it to the final result until every edge node has a fog node within its range.

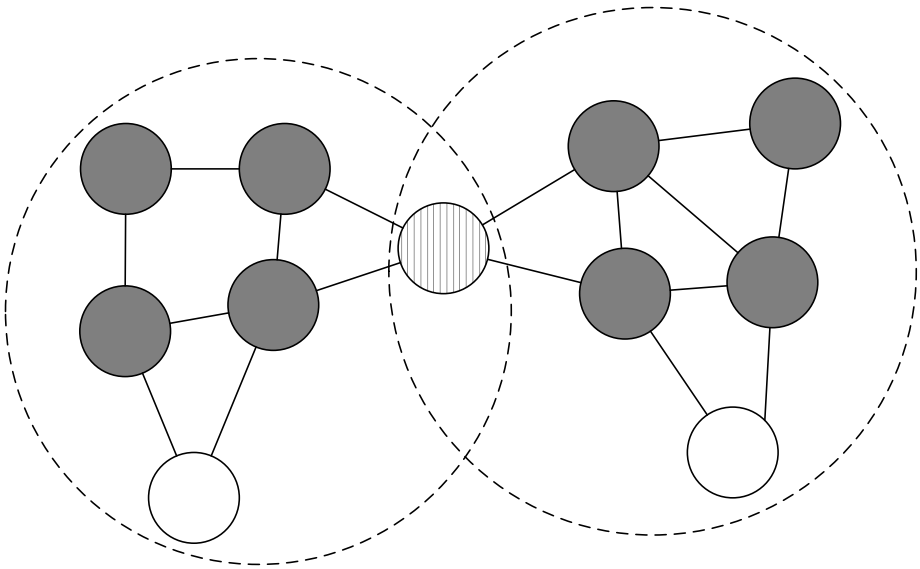
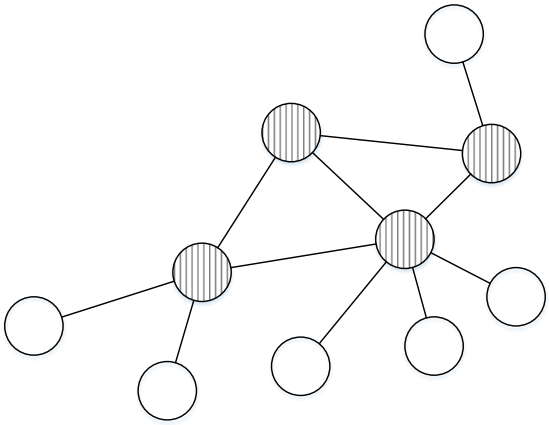


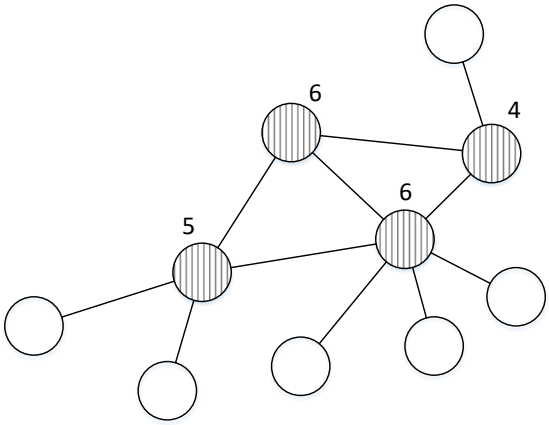
Figure 4.7: Exemplary Placement of Fog Nodes using the Threshold 2

Fig. 4.8 shows an exemplary sequence of steps placing fog nodes in a network topology. There are two type of fog nodes available: T1 with a capacity of 2 connections and costs of 1 and T2 with a capacity of 5 connections and costs of 2. Also, the threshold used is the number of hops and is limited to 2 hops. The initial network without fog nodes is depicted in (a). Shaded nodes represent the network consisting of backbone and edge nodes and white nodes represent the client devices that are connected to the network. With the first iteration, the algorithm calculates the range of each client and connects them to the respective nodes in the topology. The numbers next to the network nodes in (b) represent the number of clients that are within a 2 hop distance. Based on this number and the two fog types available, there are multiple options for the first placement. Three nodes (with 5 and 6 connections) are able to place T2 with costs of $\frac{2}{5}$ and one node (with 4 connections) with costs of $\frac{2}{4}$. Placements with T1 can only achieve costs of $\frac{1}{2}$. In case there are multiple options, the one with the minimal hop count gets picked. Therefore, a T2 fog node with costs $\frac{2}{5}$ is placed as shown in (c). The placed fog node is colored in blue and associated with the fog type T2. All covered client nodes are colored in orange. Since they do not have to be connected to another fog node, the current connection numbers are stale. For the second iteration, those numbers need to be updated. In (d) the connection numbers are updated to the one client left. For this client there are two possible placements with two possible types whereas T1 is the cheaper with costs of $\frac{1}{1}$. As before the closest placement gets chosen as shown in (e). Finally there are no clients left to connect in (f) and the algorithm terminates.

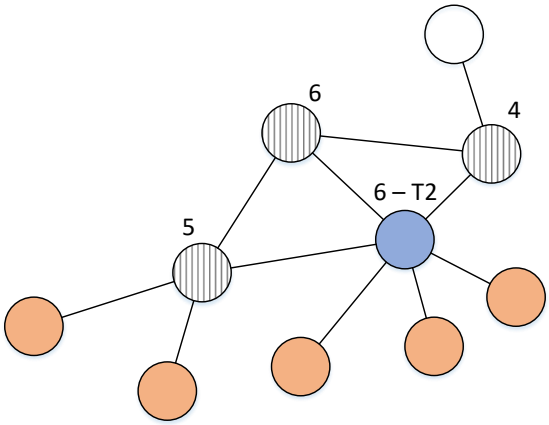
The pseudo-code of the algorithm is listed in Algorithm 4.5. Even though PLACE-FOGNODES is the main procedure to start with, there is also a second sub procedure



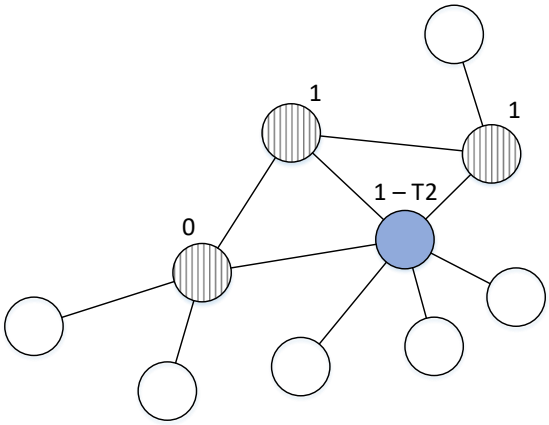
(a) Initial State



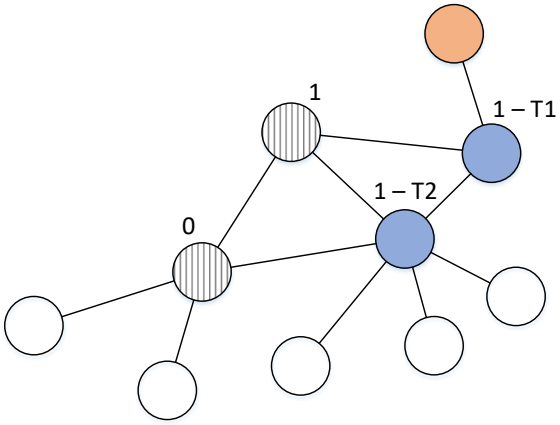
(b) First Iteration



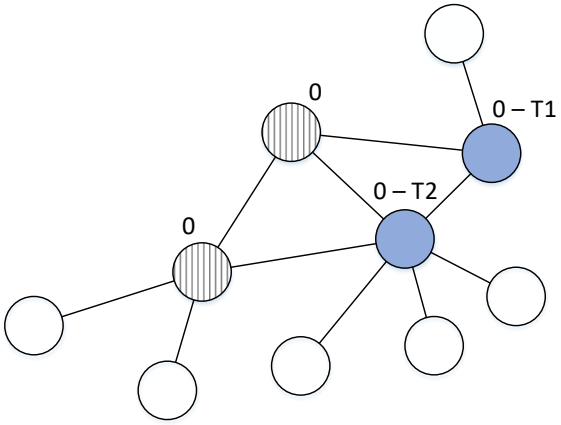
(c) First Placement with Type T2



(d) Updated Second Iteration



(e) Second Placement with Type T1



(f) Final State

Figure 4.8: Exemplary Placement of Fog Nodes in a Topology

DETERMINEPOSSIBLEFOGNODES in this listing. Each of them will be explained in more detail starting with PLACEFOGNODES.

First, PLACEFOGNODES starts with an initially empty result set of placed fog nodes F and adds in each iteration one node as a placement. Required as parameters is the graph $G = (V, E)$, the set of edge nodes $A \subseteq V$ which satisfies $A \cap B = \{\}$ as already introduced in Section 4.3 and the cost function's threshold t, e.g., the latency bound. To place fog nodes, the algorithm first needs to retrieve all available fog node placements P by calling the DETERMINEPOSSIBLEFOGNODES function, which is covered later on (line 3). As long as there are edge nodes that still need to be connected to a fog node and hence $A \neq \{\}$, a new fog node is determined. The retrieved possible nodes get sorted based on the average deployment costs per covered edge node in ascending order (line 5). In case two possible nodes have the same average deployment costs, they are compared on their average costs to connect, e.g., minimum latency. The cost optimal node is picked from the sorted set and is added to the result set F (line 6–7). All edge nodes that are covered by the determined fog node are removed from the set A to only consider uncovered edge nodes in the next iteration (line 8). Finally, the set of possible fog nodes P is updated with respect to the latest picked fog node placement by the UPDATEPOSSIBLEFOGNODES function (line 9). This function removes the covered edge node from the range of all associated nodes. As a result, the set of possible fog placements P shrinks and some nodes may have new costs associated, thus requiring to be sorted in the next iteration.

Second, the DETERMINEPOSSIBLEFOGNODES procedure takes the graph G, set edge nodes A and the cost function's threshold t as parameters. Starting from each edge node in A, the algorithm performs a Dijkstra Algorithm [Dij59] to find the best connection from this node to all others in the graph. If the costs exceeds the limit given by the threshold t parameter, the algorithm aborts, as nodes reachable after this point are irrelevant. All nodes reachable from any of the edge nodes and their respective costs are stored in the set P, making it the final outcome of this function (line 14). Starting with an initial cost of 0, each node in the graph maintains the connection costs for all edge nodes within its range (line 16). The edge node s itself gets added to the queue as the initial node to process (line 17). Each currently processed node $c \in Q$ gets added to the final set P as it is reachable from the edge of the network (line 21). To iterate the graph, the distance to each node in the direct neighborhood of the currently processed node $n \in N_1(c)$ is calculated (line 23). Only if the calculated costs are within the boundaries of the threshold t they are further considered (line 24). In case this neighbor node is already part of the queue, its costs and predecessor are updated if the connection costs are lower than the currently associated costs (line 26–29). Nodes which are not in the queue get their costs and predecessor set to the current node c and the originally starting edge node s is added to the range of this possible fog node (line 30–35). Finally, the node is added to queue to also be processed. As soon as there are no nodes left to

Algorithm 4.5 The Fog Node Placement Algorithm

```
1: procedure PLACEFOGNODES(G, A, t)
2:     F ← {}
3:     P ← DETERMINEPOSSIBLEFOGNODES(G, A, t)
4:     while A ≠ {} do
5:         SORT(P)
6:         f ← P.FIRST()
7:         F ← F ∪ {f}
8:         A ← A \ f.Range
9:         P ← UPDATEPOSSIBLEFOGNODES(P, f)
10:    end while
11:    return F
12: end procedure
13: procedure DETERMINEPOSSIBLEFOGNODES(G, A, t)
14:    P ← {}
15:    for all s ∈ A do
16:        s.Costs[s] ← 0
17:        Q ← {s}
18:        while Q ≠ {} do
19:            SORT(Q)
20:            c ← Q.DEQUEUE()
21:            P ← P ∪ {c}
22:            for all n ∈ N_1(c) do
23:                costs ← c.Costs[s] + CALCULATECOSTS(c, n)
24:                if costs ≤ t then
25:                    if n ∈ Q then
26:                        if cost < c.Costs[s] then
27:                            n.Predecessor ← c
28:                            n.Costs[s] ← costs
29:                        end if
30:                    else
31:                        n.Predecessor ← c
32:                        n.Costs[s] ← costs
33:                        n.Range ← n.Range ∪ {s}
34:                        Q.ENQUEUE(n)
35:                    end if
36:                end if
37:            end for
38:        end while
39:    end for
40:    return P
41: end procedure
```

process, all nodes within the boundaries of the threshold have been found and associated with their optimal connection costs. Hence, P is the resulting sub set of all nodes in the graph containing all reachable fog placements with costs, predecessor, and their respective range of edge nodes.

4.6 Write the Experiment Output

The final step of the tool is to write the generated network to a persistent file. This includes the assignment of devices to the edge of the network as well as the fog node placement including their respective type. Similar to the first step of reading the graph, it should be possible to extend for additional output formats. As described in Section 2.4, there are many different simulation and emulation tools that can be used for the actual measurements. Through the graph structure interface a writer can access all information tracked in the testbed.

4.7 Theoretical Complexity

The proposed workflow contains two major algorithms: the algorithm to identify the edge of the network and the algorithm to find a fog placement. In this section, the theoretical complexity of both algorithms will be examined to get indication for the running time evaluation in Section 6.

4.7.1 The Edge Identification Algorithm

The first step of the edge identification algorithm is the identification of cross-AS connections. Therefore, all edges are considered and iterated. Since the check of the associated AS and the adding process to the set B can be achieved in $\mathcal{O}(1)$, this step will be processed in $\mathcal{O}(|E|)$.

In the second step, the average degree has to be calculated which is possible in $\mathcal{O}(|V|)$ as each node has to be visited once. With the average degree calculated, all remaining nodes that are not part of the backbone $V \setminus B$ are iterated. In the worst case, this set equals V and hence it can be done in $\mathcal{O}(|V|)$.

The third and last step is based on the Breadth-First algorithm, which in its standard form has a proven complexity of $\mathcal{O}(|V| + |E|)$ [CLRS09]. Compared to the original algorithm the only thing changed is the iteration back to connect the backbone partitions

(line 16–22). Since this operation cannot revisit an edge twice, the worst case would be to visit each once and hence leads to $\mathcal{O}(|E|)$. In total, this would be a complexity of $\mathcal{O}(|V| + |E| + |E|)$ for the third step.

Combining the complexity of all three steps and considering that they run sequentially one after another, it can be seen that the total complexity of the edge identification algorithm is $\mathcal{O}(|E| + |V| + |V| + |E|) = \mathcal{O}(|V| + |E|)$.

4.7.2 The Fog Node Placement Algorithm

In the PLACEFOGNODES function, the first relevant operation is the call of the DETERMINEPOSSIBLEFOGNODES function, which will now be discussed in more detail. The Dijkstra algorithm used for the calculation of the best connection has a proven complexity of $\mathcal{O}(|E| \cdot T_{dk} + |V| \cdot T_{em})$ where T_{dk} is the complexity of the *decrease-key* and T_{em} the complexity of the *extract-minimum* operations [CLRS09]. Therefore, the complexity depends on the chosen structure for the queue Q. The overall complexity also depends on the complexity T_{cc} of the CALCULATECOSTS function called on each edge of the first neighborhood for the currently processed vertex. For most cost functions like latency, hop, bandwidth, etc., the costs can be calculated in $\mathcal{O}(1)$. As an edge can only be visited twice, once for each end point, the number of calls of the function CALCULATECOSTS is at most $2|E|$. In total this leads to $\mathcal{O}(|E| \cdot (T_{dk} + T_{cc}) + |V| \cdot T_{em})$.

With the possible fog node placements in P, the PLACEFOGNODES function now can select the currently optimal placement until there is nothing left to connect to (line 4–10). Since it is possible that $A = V$, the while loop iteration count is based on the set V. Sorting a set of n elements can be done within a complexity of $\mathcal{O}(n \cdot \log(n))$, for instance by the use of the merge-sort algorithm. Removing the range of the chosen fog node f from the set of edge nodes can be done in $\mathcal{O}(|V| \cdot \log(|V|))$, as every edge node in A can only be removed once from the set A and by the use of a tree structure it is possible to remove an element in $\mathcal{O}(\log(|V|))$. The update of the remaining fog node placements with the UPDATEPOSSIBLEFOGNODES function has a complexity of $\mathcal{O}(|V|)$ because every remaining node in P gets updated at most once. All of the above leads to a complexity of $\mathcal{O}(|E| \cdot (T_{dk} + T_{cc}) + |V| \cdot T_{em} + |V| \cdot \log(|V|) + |V| \cdot (|V| \cdot \log(|V|) + |V|)) \subseteq \mathcal{O}(|E| \cdot (T_{dk} + T_{cc}) + |V|^2 \cdot \log(|V|))$.

5 Implementation

To evaluate and make use of the concept presented in this work, it has been implemented in the *EmuFog*[1] tool. It is published open source on the GitHub platform under the MIT license to make it usable and expendable for everyone. Since it is written in the Java programming language, it is platform-independent and hence can be easier combined with new topology generators and emulators.

This implementation section is structured as follows. First, all mandatory and optional input files and parameters are explained in Section 5.1. The generated output format is described in Section 5.2. Section 5.3 explains how to run EmuFog from the command line, and the last Section 5.4 presents the implementation details on the package level.

5.1 Input Data

In this Section all necessary input files will be discussed. All of them are required at the local file system and will be read at execution time. None of them get modified or overwritten during the execution. They can be passed as arguments to EmuFog via the command line interface explained in Section 5.3.

5.1.1 The Network Topology

The first input to consider is the network topology to operate on. As described in Section 4.2, it should be possible to use different input formats. Currently, EmuFog supports the BRITE and the Caida data format. Besides existing topologies stored in data files, it is also possible to extend EmuFog and create a network topology programmatically with the offered interface of the Graph package (discussed in Section 5.4.4). In this case, there is no need for a topology file as an input.

[1] EmuFog is available at https://github.com/emufog/emufog

For BRITE it only requires a single file with the *.brite* extension. This data format is independent of the model used to generate the topology. Even though the BRITE generator supports multiple output formats, the reader currently implemented only supports the *.brite* format. Besides the nodes and edges of the topology, it also reads the latency and bandwidth generated by BRITE.

In contrast to BRITE, Caida requires multiple files to build a topology. Caida's *Internet Topology Data Kit*[2] (ITDK) is split into pieces. To combine and build the network, the reader requires the geographical information of the nodes in the *.nodes.geo* file, the mapping of nodes to the autonomous system in the *.nodes.as* file and the links between nodes in the *.links* file. In contrast to BRITE, there are not latency and bandwidth information available hence they have to be estimated based on the geographical information provided.

5.1.2 The Settings File

Besides the network topology, the settings file as another mandatory input for EmuFog. It contains additional properties affecting the placement and performance as well as the definition of hardware types and the Docker image to run. With the start of EmuFog, the settings will be read from the hard drive and applied to the execution. Listing 5.1 shows the settings file with exemplary values. To make it easier to read and understand for humans, the chosen data format is JSON. This also makes it possible to create structures needed to model the hardware containers.

The "DeviceNodeTypes" lists all device that will be placed automatically in the network. As devices and fog nodes both base on Docker container to emulate different hardware setups the structure is similar to another. With the use of the scaling factor devices can simulate lots of smaller devices as sensors in a single container. In comparison to the devices, fog nodes have a maximal connections property to limit the number of connections for low resource hardware. Since the goal is to minimize deployment costs they have to be present. The costs property therefore only affects fog nodes. Each property in the settings file is described in Table 5.1.

5.2 Output Files

After placing fog nodes in the network the final network topology will be exported to run an experiment. Since one of the criteria of EmuFog is the extensibility, it should

[2]Internet Topology Data Kit: https://www.caida.org/data/internet-topology-data-kit/

```
{
  "BaseAddress": "10.0.0.0",
  "OverWriteOutputFile": true,
  "MaxFogNodes": 100,
  "CostThreshold": 2,
  "HostDeviceLatency": 0,
  "HostDeviceBandwidth": 1000,
  "ThreadCount": 1,
  "ParalleledFogBuilding": false,
  "DeviceNodeTypes": [
    {
      "DockerImage": {
        "Name": "ubuntu",
        "Version": "latest"
    },
    "ScalingFactor": 1,
    "AverageDeviceCount": 1,
    "MemoryLimit": 524288000,
    "CPUShare": 1
    }
  ],
  "FogNodeTypes": [
    {
      "ID": 1,
      "DockerImage": {
        "Name": "ubuntu",
        "Version": "latest"
      },
      "MaximumConnections": 1,
      "Costs": 1,
      "MemoryLimit": 1048576000,
      "CPUShare": 1
    },
    {
      "ID": 2,
      "DockerImage": {
        "Name": "debian",
        "Version": "stable"
      },
      "MaximumConnections": 5,
      "Costs": 2.5,
      "MemoryLimit": 2097152000,
      "CPUShare": 1.5
    }
  ]
}
```

Listing 5.1: An exemplary Settings File

Parameter	Type	Description
BaseAddress	Text	The base IP address of the network containing all nodes. This is the starting point and first address assigned. Format: *XXX.XXX.XXX.XXX*
OverWriteOutputFile	Boolean	Indicates whether the output file should be overwritten in case it already exists. *true* to overwrite file, *false* to keep it.
MaxFogNodes	Integer	The maximum number of fog nodes to place in the network.
CostThreshold	Float	The cost function's threshold. Depends on the cost function chosen.
HostDeviceLatency	Float	Latency to use between a placed client device at the edge of the network and its associated edge node.
HostDeviceBandwidth	Integer	Bandwidth to use between a placed client device at the edge of the network and its associated edge node. Measured in MB.
ThreadCount	Integer	Number of threads to use for processing the graph. Each AS is processed by an individual thread.
ParalleledFogBuilding	Boolean	Indicates whether the fog network is built in parallel. *true* to build it up in parallel, *false* to build it sequentially.
DeviceNodeTypes	List	A list of all device types to assign to edge nodes.
ID	Integer	Unique identifier of a computing container.
DockerImage	Object	Docker image associated with this container.
Name	Text	Name of the Docker image to use.
Version	Text	Version of the Docker image.
ScalingFactor	Float	Scales the workload of this device higher than 1.
AverageDeviceCount	Float	The average number of devices of this type connected to an edge node.
MemoryLimit	Integer	Memory size of this container in Bytes.
CPUShare	Float	Scaling of the CPU power. Container receives share of its value in respect to the total sum.
MaximumConnections	Integer	The maximum number of connections from client devices this container can handle.
Costs	Float	Deployment costs of this fog node in the network.

Table 5.1: The Settings File Parameter in Detail

be possible to extend an output to any data format as long as it can be build with the information provided in the graph. The current implementation includes an export function to the MaxiNet emulator introduced in Section 2.4.1. Listing 5.2 shows the output file generated by an exemplary execution on a network topology of 10 nodes. First, the nodes running actual software are added to the topology (line 13–16). In this case, those are two edge nodes (r38 and r37) and two devices (h0 and h1) with their respective IP address, memory limit and the Docker image to use. Second, all remaining nodes that do not run software are added (line 19–26 & 29). Now all nodes in the topology are added, but not connected yet. Third, all links between nodes are added with their latency delay and bandwidth (line 32–54). Finally, the topology gets initiated and the experiment starts (line 57–59).

5.3 How to use EmuFog

The first step to launch EmuFog is to gain any type of network topology. As mentioned before, this could be an artificial one by any topology generator like in Section 2.3.2 or a measured snapshot of the actual Internet from research projects introduced in Section 2.3. Second, it requires a settings file specifying hardware types, limits, threads etc. as previously explained in detail in Section 5.1.2.

For an exemplary launch of EmuFog, it requires a compiled version `emufog.jar` available as well as a settings file `settings.json` and a topology generated by BRITE `topology.brite`. All input files are located in the same directory as the binary to minimize this example. Those inputs have to be passed to EmuFog by its command line interface exemplary shown in Listing 5.3. The full list of arguments is provided in Table 5.2, including their respective description.

```
$ java -jar emufog.jar -s settings.json -t brite -f topology.brite -o out.py
```

Listing 5.3: An exemplary Launch of EmuFog

5.4 EmuFog Structure

As previously mentioned, EmuFog is written in Java and uses features of the Java 8 standard. Hence, it requires the Java Development Kit (JDK) 8 and up to compile and execute the software. To ease the building of EmuFog including its dependencies, the project uses the Gradle build tool. It has been developed with the currently latest version

```
1  #!/usr/bin/env python2
2
3  import time
4
5  from MaxiNet.Frontend import maxinet
6  from MaxiNet.Frontend.container import Docker
7  from mininet.topo import Topo
8  from mininet.node import OVSSwitch
9
10 topo = Topo()
11
12 # add hosts
13 r38 = topo.addHost("r38", cls=Docker, ip="10.0.0.4", dimage="ubuntu:trusty", mem_limit=1048576000)
14 s37 = topo.addHost("s37", cls=Docker, ip="10.0.0.3", dimage="ubuntu:trusty", mem_limit=1048576000)
15 h0 = topo.addHost("h0", cls=Docker, ip="10.0.0.1", dimage="ubuntu:latest", mem_limit=524288000)
16 h1 = topo.addHost("h1", cls=Docker, ip="10.0.0.2", dimage="ubuntu:latest", mem_limit=524288000)
17
18 # add switches
19 r39 = topo.addSwitch("r39")
20 s32 = topo.addSwitch("s32")
21 s33 = topo.addSwitch("s33")
22 s34 = topo.addSwitch("s34")
23 s35 = topo.addSwitch("s35")
24 s36 = topo.addSwitch("s36")
25 s30 = topo.addSwitch("s30")
26 s31 = topo.addSwitch("s31")
27
28 # add connectors
29 c0 = topo.addSwitch("c0")
30
31 # add links
32 topo.addLink(s30, s32, delay='2.8683429ms', bw=10.0)
33 topo.addLink(s31, s32, delay='1.8749269ms', bw=10.0)
34 topo.addLink(s33, s30, delay='2.7793577ms', bw=10.0)
35 topo.addLink(s35, s30, delay='3.6889384ms', bw=10.0)
36 topo.addLink(s32, s34, delay='0.52105445ms', bw=10.0)
37 topo.addLink(s35, s33, delay='0.91326535ms', bw=10.0)
38 topo.addLink(s31, s34, delay='2.3148987ms', bw=10.0)
39 topo.addLink(s30, s34, delay='3.0580752ms', bw=10.0)
40 topo.addLink(s37, s34, delay='2.3839264ms', bw=10.0)
41 topo.addLink(r38, s30, delay='3.511378ms', bw=10.0)
42 topo.addLink(s33, s34, delay='1.1840962ms', bw=10.0)
43 topo.addLink(s37, s36, delay='0.35953817ms', bw=10.0)
44 topo.addLink(s36, s35, delay='1.5631313ms', bw=10.0)
45 topo.addLink(s34, s35, delay='1.5446935ms', bw=10.0)
46 topo.addLink(s36, s30, delay='2.8834488ms', bw=10.0)
47 topo.addLink(s31, s30, delay='1.7750531ms', bw=10.0)
48 topo.addLink(r39, s37, delay='1.3851869ms', bw=10.0)
49 topo.addLink(r39, s30, delay='1.3370553ms', bw=10.0)
50 topo.addLink(s33, s36, delay='1.2379903ms', bw=10.0)
51 topo.addLink(s37, s30, delay='2.5305889ms', bw=10.0)
52 topo.addLink(r38, c0, delay='0.0ms', bw=1000.0)
53 topo.addLink(c0, h0, delay='0.0ms', bw=1000.0)
54 topo.addLink(r39, h1, delay='0.0ms', bw=1000.0)
55
56 # create experiment
57 cluster = maxinet.Cluster()
58 exp = maxinet.Experiment(cluster, topo, switch=OVSSwitch)
59 exp.setup()
```

Listing 5.2: An exemplary Experiment for MaxiNet

Argument	Shortcut	Description
-Settings	-s	Path to the settings file to use.
-Type	-t	The type of reader to use. Currently supported: *BRITE* and *CAIDA*. This argument is case insensitive.
-Output	-o	Path to the output file to write.
-File	-f	Path to a topology file to read in. This argument can be used multiple times.

Table 5.2: Command Line Arguments for EmuFog

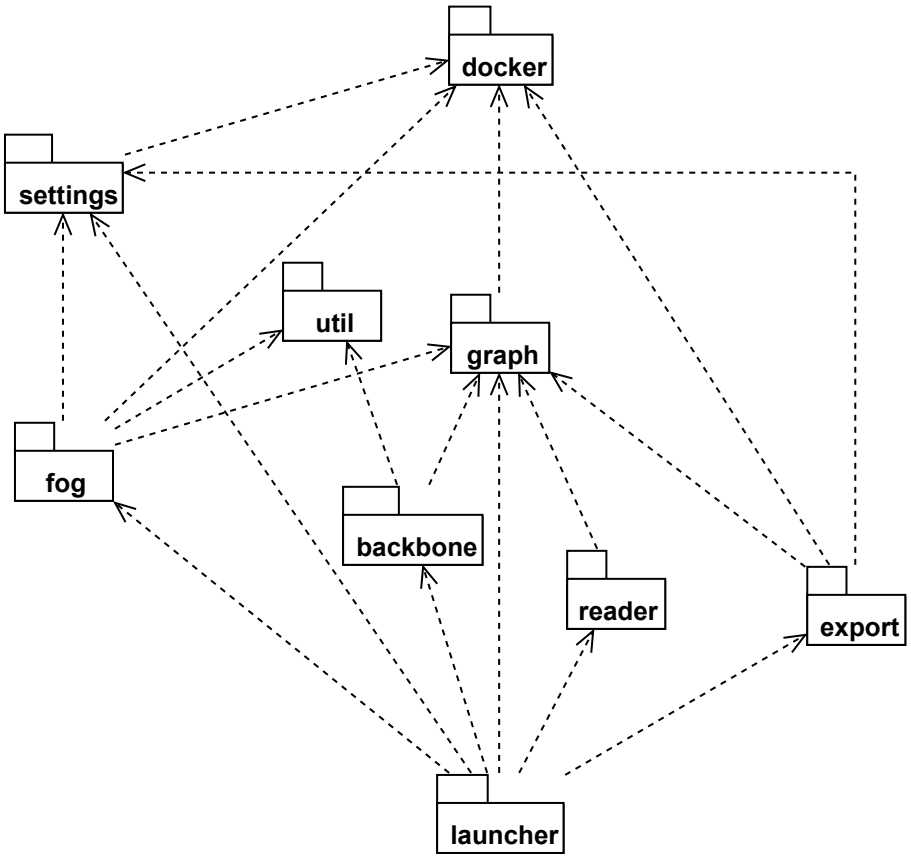


Figure 5.1: The Package Diagram of EmuFog

of Gradle 3.4. The overall functionality of EmuFog is broken down into several modules. Fig. 5.1 provides an overview of the EmuFog's structure using a package diagram.

In the following sections, each package will be discussed in more detail to explain its functionality and its most important classes and interfaces. Due to the hierarchy of the dependencies, packages with already covered dependencies will be explained first.

5.4.1 The Docker Package

This package contains the hardware specification as well as the software to run with the emulation tool of choice. EmuFog uses Docker containers to emulate the hardware and software to evaluate. Docker is available cross-platform, supports hardware resource limitations to emulate different devices, and is a lightweight option for an isolated environment to execute code. This is crucial as multiple containers will run on the same physical machine thus making isolation a necessity. Also Docker offers a wide range of preexisting containers than can either be used directly or modified easily to include any software of choice.

The abstract `DockerType` class models a generic Docker container with the amount of memory, its share of the CPU and the image to run. Fog nodes as well as devices in the network rely on such contains and hence there are two classes `FogType` and `DeviceType` respectively. They model the different nodes with some additional attributes, but extend the generic `DockerType` class.

5.4.2 The Settings Package

The settings package consists of two classes: `Settings` and `SettingsReader`. The latter provides a function to read in the settings from the setting file explained in Section 5.1.2 passed via the command line. To transform the JSON document into a Java object, EmuFog uses the open source Gson[3] library by Google. It can transform basic types as well as structured objects as the hardware specifications. The `Settings` class itself holds all the provided information easy to access for all other packages.

5.4.3 The Util Package

In comparison to the other packages, the util package does not provide features explicitly tailored for the actual testbed. The main component in this package is the `Logger` class offering a centralized logging system using the singleton pattern. Different logging entries can categorized with the `LoggerLevel` enum into `INFO`, `WARNING`, and `ERROR` making it easier to recognize them in the output. All entries are stored and can be optionally written into a text file.

[3] Gson is available at https://github.com/google/gson

5.4.4 The Graph Package

Independent of the input format, all network topologies are stored in a simplified graph structure based on the model explained in Section 4.1. Each node in the graph is an instance of the abstract `Node` class. Possible nodes are `HostDevice` for devices, `Router` for edge nodes, and `Switch` for backbone nodes. A node has a unique identifier and a list of `Edge` instances connecting it to other nodes in the graph. Those nodes belong to an instance of the `AS` class storing a mapping of node identifiers to the actual node objects.

The graph packages offers two graph representations: the `Graph` and the `CoordinateGraph` class. First, the `Graph` class contains all of its associated autonomous systems which hold the actual nodes of the graph. Besides that, it offers the interface to build a graph to other packages as the reader package. This includes functions to create the different kinds of nodes as well as edges in the graph. To create an edge between two nodes in this class, it is required to know the latency between them. Even though this is often no problem, there are datasets not providing this information as they only model the topology not the characteristics of the edges. Therefore, there is a second graph presentation with the `CoordinateGraph` class which extends the original `Graph` class. It extends the set of functions by the option to create nodes and associate them with coordinates for future use. Creating an edge accepts in this class an instance of the `ILatencyCalculator` interface having a single function to calculate the latency based on the x and y coordinates in the two dimensional plane. This way new data format readers can either reuse existing calculators or simply implement their own to open up for all types of networks. There is lots of research [DCKM04; GSG02; LHC03; NZ02; SPPS08; SXBL06] in the field of estimating latencies by the use of coordinate systems.

5.4.5 The Reader Package

Depending on the data format to read in, there has to be respective reader available. Currently this package contains two readers: the `BriteFormatReader` and the `CaidaFormatReader`. For future use, new readers can be added by extending the abstract `GraphReader` class. The main launcher of EmuFog will call the reader and expects a graph instance in return. To build a graph the interface of the previously mentioned `Graph` class has to be used.

5.4.6 The Backbone Package

The identification of the edge of the network is the second step in the overall workflow. In contrast to the intuitive assumption, EmuFog does not identify the edge but rather identifies the backbone of the network. All that is left is the edge of the network. The backbone package contains the algorithm to detect the edge presented in Section 4.3.

An identification of the backbone can be triggered via the `BackboneClassifier` class by passing the graph object to operate on. Since different autonomous systems can be processed independently, they get parallelized; thus, speeding up the identification on multi-AS topologies. Each AS is mapped to its own `BackboneWorker` class identifying only the associated one. All workers are executed in parallel by the use of a thread pool with the given numbers of threads to use by the settings file.

5.4.7 The Fog Package

The fog package implements the fog node placement algorithm introduced in Section 4.5. Similar to the backbone identification, the `FogNodeClassifier` class takes the graph to process and splits the task into pieces of one AS each. Each of those tasks will be executed in its own thread with the abstract `Worker` class. Depending on the ”ParalleledFogBuilding” parameter in the settings file, this is either a `SequentialFogWorker` or a `ParallelFogWorker`.

5.4.8 The Export Package

After determining the fog nodes, the results of the network have to be exported from EmuFog to an emulator of choice. Therefore, it has to be expendable to new export classes which is possible by implementing the `IGraphExporter` interface consisting of a single function to export a graph structure. Currently, EmuFog supports an export to MaxiNet with the `MaxiNetExporter` class implementing the interface. It generates the Python file and writes it to a given path.

5.4.9 The Launcher Package

Finally, there is the launcher package; this includes the `Emufog` class that launches the entire application. It executes all steps depicted in Fig. 4.1 in this sequence. As a result, it has dependencies to nearly all of the previously covered packages combining them

to build EmuFog. The main class `EmuFog` offers a command line interface parsed by the jCommander[4] library. Additional properties are passed via the settings file.

[4]jCommander is available at http://jcommander.org/

6 Evaluation

The evaluation section provides different evaluations of EmuFog and MaxiNet. First, Section 6.1 measures the running time of the edge identification algorithm and the fog placement algorithm. Second, as the fog node placement has a cost optimal result the quality of the heuristic presented can be measure. Section 6.2 compares the results of the heuristic with the optimal solution to provide indication of the approximation factor. The last evaluation in Section 6.3 measures the deployment time of EmuFog outcomes launched with MaxiNet.

6.1 Running Time Measurements

In the first test of the evaluation, the running time of EmuFog gets evaluated. To show the versatile nature of EmuFog, we use two different Internet topology datasets: first, a synthetic topology generated by the BRITE topology generator, using the model of Albert and Barabási [AB00]; second, a real world topology from Caida[1] measured in 2014. A detailed description of all networks used is listed in Appendix A.

For the performance evaluation, we used autonomous systems of different sizes (n = 10, 100, 1,000 and 10,000 nodes). Each size is evaluated with five different samples and ten runs each. For the BRITE dataset, the autonomous systems are generated with exactly n nodes. From the Caida dataset, we select autonomous systems with a deviation of $\pm 10\%$ from n so that enough different autonomous systems of similar size can be found. The evaluation was carried out on an Intel i5-4670K processor with 4 physical and logical threads @3.4GHz using 16 GB RAM; the operating system was Ubuntu 17.04.

We implemented adapters for both datasets to generate a generalized topology for edge identification and fog node placement. In the evaluations, we measure the latency for performing the two major preprocessing steps in EmuFog: Edge identification in Section 6.1.1 and fog node placement in Section 6.1.2.

[1] Caida ITDK 2014-12: http://data.caida.org/datasets/topology/ark/ipv4/itdk/2014-12/

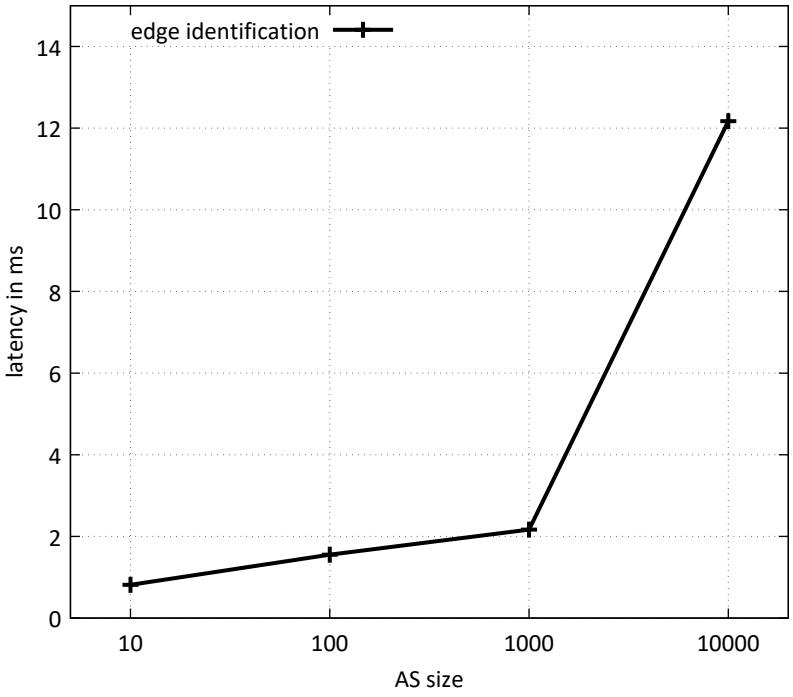


Figure 6.1: Edge Identification for the BRITE Data Set

6.1.1 Edge Identification

The performance evaluations starts with the edge identification, the logical necessary step before the fog placement. Fig. 6.1 shows the running time of the edge identification for the BRITE dataset and Fig. 6.2 shows the identification for the Caida Data Set. On the x-axis are the AS sizes displayed on a logarithmic scale and on the y-axis the running time in ms on a linear scale.

For the BRITE dataset, it can be seen that the time taken increases strongly monotonic with the size of the AS to process. But it is also visible that even though the growth for autonomous systems up to 1,000 nodes is fairly slow, it increases strongly for big networks. As previously explained in Section 4.7.1, the performance of the algorithm does not only depend on the number of nodes, but also on the number of edges. This leads to a non linear characteristic of the running time.

The results of the Caida dataset look different from the ones of the BRITE dataset. In comparison, it is visible that the time taken does not increase monotonic but instead has a drop for autonomous systems with the size of approx. 1,000 nodes. Also, the span between the longest and the shortest running time is is smaller than in the BRITE dataset. This might be due to the fact that the topologies in the Caida dataset are partitioned;

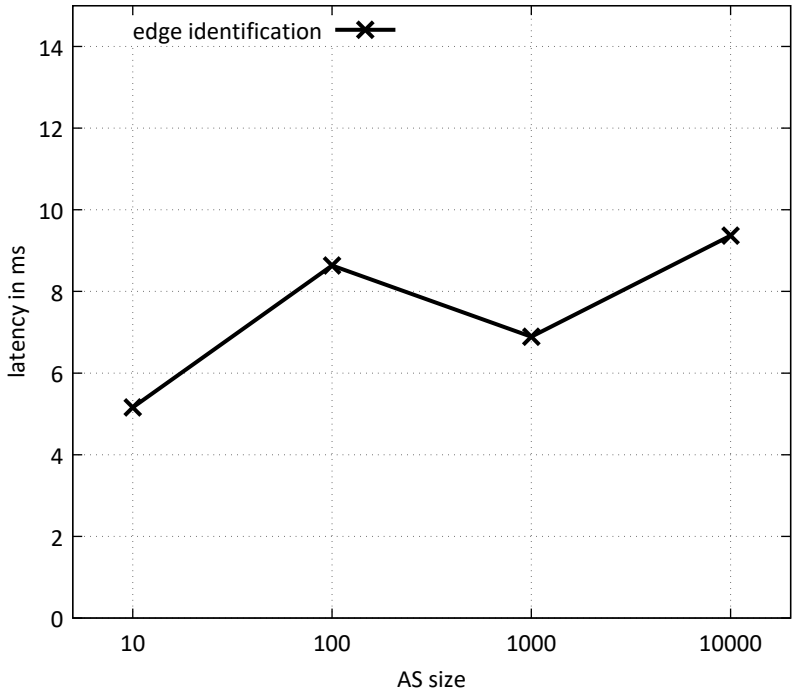


Figure 6.2: Edge Identification for the Caida Data Set

hence the algorithm has to process multiple times smaller sub partitions of the network instead of a single execution.

Finally, it can be concluded that an average running time of 12.171ms for an AS with the size of 10,000 nodes is sufficiently fast and does not limit the usage of EmuFog in any way. Even the biggest autonomous system of the real world Caida data set with approx. 450,000 nodes is processed in less than a second.

6.1.2 Fog Node Placement

Compared to the edge identification, the fog node placement is a more complex problem in terms of theoretical complexity as well as its evaluation. Again, the evaluation is carried out on both datasets. As the fog placement depends on multiple parameters in the settings file, the evaluation had to be done with different settings. Therefore, two different fog types were available: a small fog node with a capacity of 5 connections and deployment costs of 2 and a bigger node with a capacity of 25 connections and deployment costs of 5.5. To evaluate the placement and its dependency on the cost function's threshold t, each AS gets processed with $t = 1, 2, 4, 8, 16$. The respective cost function is the aggregated latency of connections.

The results for the BRITE dataset are depicted in Fig. 6.3. In the case of the BRITE dataset, the necessary latency is provided by the model itself. First, all diagrams show a monotonic increase with an increasing AS size. Second, the impact of the threshold t is clearly visible throughout all diagrams. An increasing threshold affects the running time negatively if the topology is too large. This is best visible with the topologies of the size 10,000. Looking at $t = 1, 2, 4, 8$ the absolute time always increases with an increasing threshold. For $t = 8, 16$ there is barely any change in the numbers. When this happens, a higher threshold will not affect the running time any more, since the threshold exceeds the diameter of the graph. This is also visible for the AS size 10 where there is no change after $t = 2$.

Fig. 6.4 shows the results of the Caida dataset. Equally to the BRITE dataset, all running times increase monotonic throughout all tested thresholds. Also, the exceeding of the diameter is visible for the samples discussed in the BRITE dataset. Hence, the comparison diagram looks similar in both datasets. However, the running time of the Caida dataset is slightly higher. Since the number of nodes is nearly the same, the increase results of the structure of the network. The number of edge in the Caida dataset is lower and the network partitioned. The partitioning of networks in the Caida datasets leads to more placements of fog nodes as they can not cover as many nodes as they could.

6.2 Quality Measure

In the previously presented evaluations, the only criteria was the performance in terms of execution time. Even though the execution time is a crucial criteria for an approximation, especially compared to the time it takes to solve it optimally, it does not consider the quality of the solution at all. To complete the evaluation of the algorithm proposed, this section formulates the problem of fog node placement in an integer linear program (ILP) and solves datasets with an ILP solver to evaluate the approximation factor.

6.2.1 Problem Formulation

For the ILP formulation of the fog node placement problem, there are various variables to be defined. As already defined, the problem is based on a graph $G = (V, E)$ with the vertex set V and the edge set E, where the vertex is separated in a set of backbone nodes $B \subseteq V$ and a set of edge nodes $A \subseteq V$ satisfying $A \cap B = \{\} \wedge A \cup B = V$. Each node $v \in V$ in the graph has a boolean variable $x_v \in \{0, 1\}$ representing if this node v is a fog node, is equal to 1, or not. Since every edge node has to have a fog node within the given range, the connections have to be modeled in the ILP as well. Therefore, exists

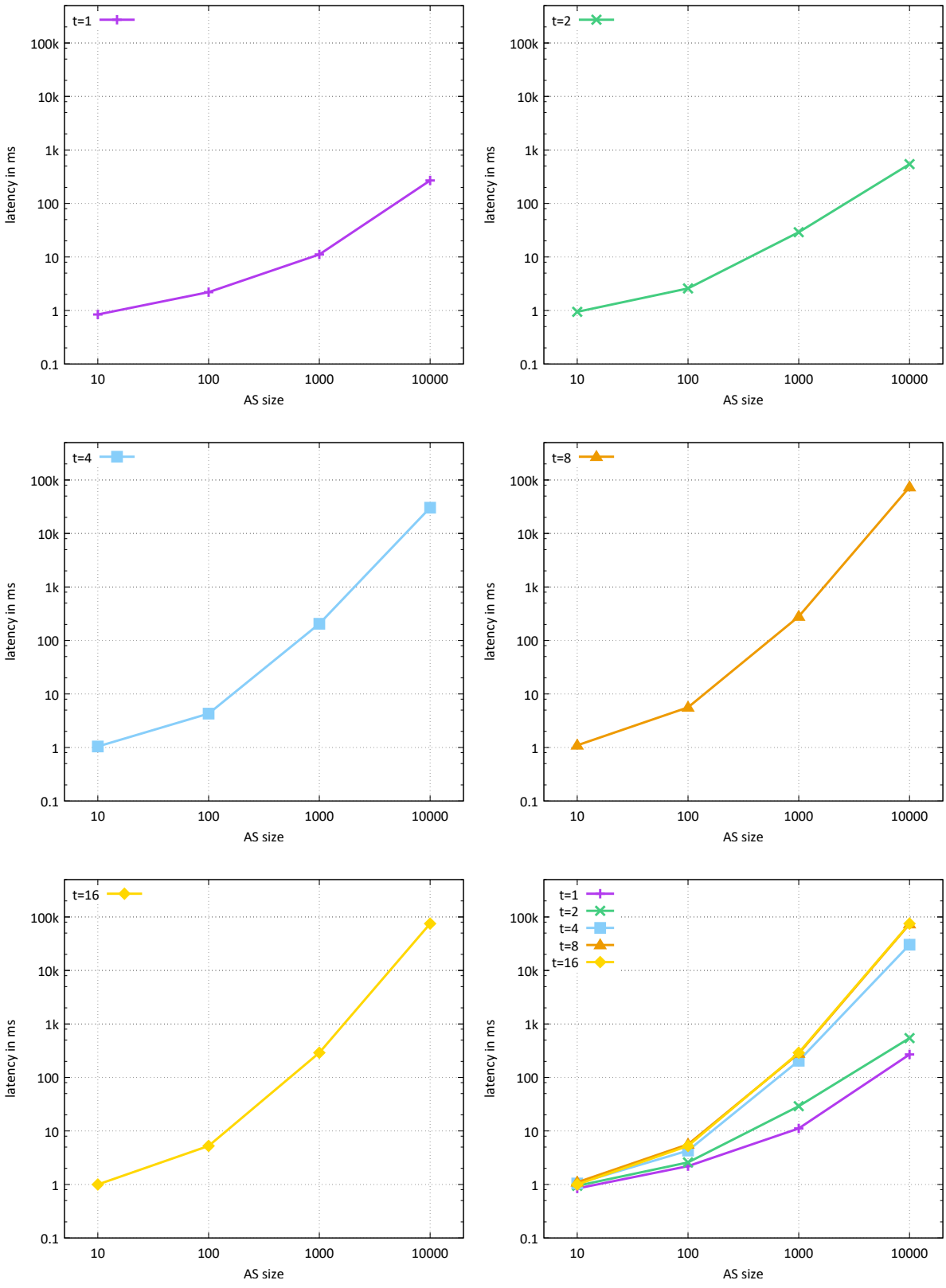


Figure 6.3: EmuFog Running Times on the BRITE Data Set

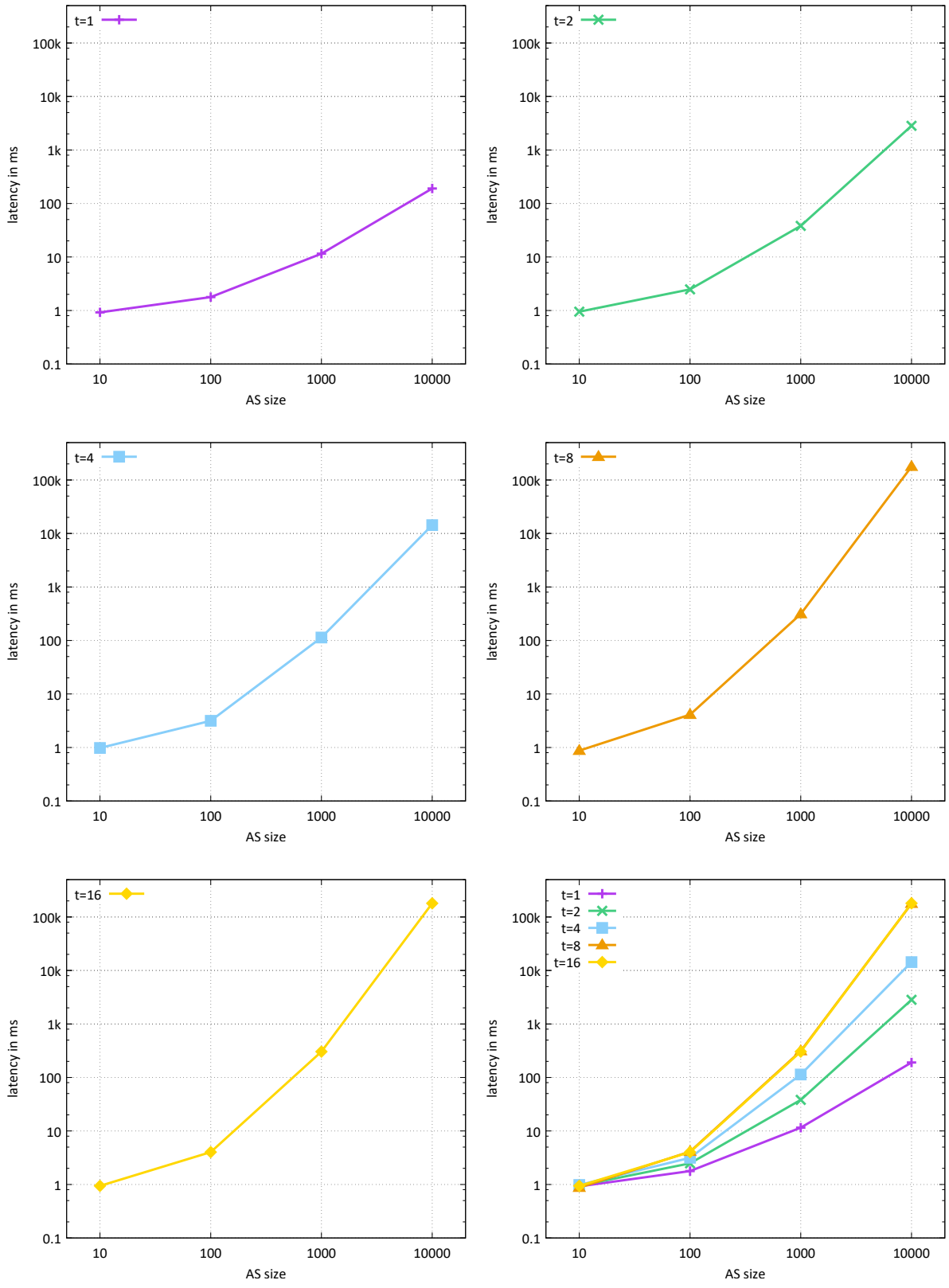


Figure 6.4: EmuFog Running Times on the Caida Data Set

the boolean connection variable $c_{a,b} \in \{0, 1\}$ representing a connection from $a \in V$ to $b \in V$. Each connection has a certain latency in the network. This latency is modeled by $l_{a,b}$ for the each connection from $a \in V$ to $b \in V$. It is required to be a positive real number $l_{a,b} \in \mathbb{R},\ l_{a,b} \geq 0$.

Aside from the connections and latencies, there is also the problem of choosing an optimal fog node type. Hence, there is a set T of all available fog node types. Each type $t \in T$ is available for each possible node $v \in V$ and therefore it is modeled as $t_v \in \{0, 1\}$ representing if the node v is using the type t. The deployment costs of a fog node type $t \in T$ is modeled by d_t with $d_t \in \mathbb{R},\ d_t \geq 0$. Additionally to the deployment costs, each fog type is also associated with a maximum capacity of connections it can handle. This capacity is modeled by cap_t for each type $t \in T$ with $cap_t \in \mathbb{N},\ cap_t \geq 0$.

The objective function to optimize is the minimization of the sum of the deployment costs d_t of all selected fog node types. In case the ILP has multiple results minimizing the objective function it can further be optimized by minimizing a second objective function only using the set of possible results minimizing the first objective function. Since the first objective already minimizes the deployment costs to a minimum, it can further be optimized by minimizing the latency from the edge to the fog. This second level order has also been implemented in the fog node placement algorithm presented in Section 4.5.2.

Apart from the constraints for binary variables, there are three additional constraints to express the problem of fog node placement. The first constraint guarantees that every edge node $a \in A$ is connected to at least one fog node. This means at least one connection leaves a and therefore, the respective identifier is 1. The second constraint guarantees that every chosen fog node only has one fog type associated. Therefore, the sum of all available fog types has to be equal to the boolean identifier whether that node is a selected fog node. In case the node is selected as a fog node, x_v is 1 and so has to be the sum of fog type identifiers. This also makes sure that there is exactly one fog type associated. If no fog type is selected, the respective node can not be a selected fog node either. The third constraint guarantees that the number of connections to a fog node do not exceed the capacity limits of the available fog node types. To achieve this the sum of connections pointing to the current node $v \in V$ must be less than the sum of the capacities of the available fog node types cap_t. In combination with the second constraint, the second sum can only have one summand.

The fog node placement problem is formulated as an integer linear program, as stated below.

$$
\begin{aligned}
& \sum_{v \in V} x_v \sum_{t \in T} d_t t_v \rightarrow \min_{x_v, v \in V}, \quad \sum_{a \in A} \sum_{v \in V} l_{a,v} x_v \rightarrow \min \\
s.t. \quad & \sum_{v \in V} c_{a,v} \geq 1 && \forall a \in A \\
& \sum_{t \in T} t_v - x_v = 0 && \forall v \in V \\
& \sum_{s \in V} c_{s,v} - \sum_{t \in T} cap_t t_v \leq 0 && \forall v \in V \\
& x_v \in \{0,1\} && \forall v \in V \\
& c_{a,b} \in \{0,1\} && \forall a, b \in V \\
& t_v \in \{0,1\} && \forall t \in T, \forall v \in V
\end{aligned}
$$

6.2.2 Approximation Results

This evaluation compares the total deployment costs of the fog placement found by the greedy algorithm presented with the optimal placement calculated by the ILP. Therefore, the oj! Algorithms[2] library is used to generate an ILP directly from the graph structure in EmuFog. Due to the time it takes to compute an optimal solution, it is impossible to use the same topologies used in the time measurements, Section 6.1. For this evaluation AS sizes of 25, 50, 75 and 100 nodes have been used. They got generated by the BRITE generator using the Albert and Barabási model [AB00].

As it would require a mathematical proof, this evaluation can not prove the quality of an approximation. Rather it can show indication of the quality based on small networks. There might be higher approximation factors than the numbers presented in this section as this depends on the structure of the graph, the capacity and costs of the placeable fog nodes, and the cost function's threshold, etc.

To evaluate the topology samples, two different configurations are used. They use fog types of different sizes to evaluate the effect of covering different numbers of edge nodes at the same time. Both use two different types to offer possibilities for misplacement and a higher total cost. Configuration 1 uses a node with a capacity of 1 and costs 1 and a node with capacity 5 and costs 2.5. Configuration 2 uses a node with a capacity of 5 and costs 2 and a node with capacity 25 and costs 5.5. Both use the cost function threshold t of 2 where the cost function is the total latency. For each AS size 5 different topologies have been run 5 times each; providing 25 samples per AS size.

Fig. 6.5 depicts the average ratio $r = \frac{costs_{greedy}}{costs_{ilp}}$ of the greedy and the optimal ILP costs for both configurations. Obviously r can never be less than 1 as in this case the greedy

[2] oj! Algorithms is available at http://ojalgo.org/

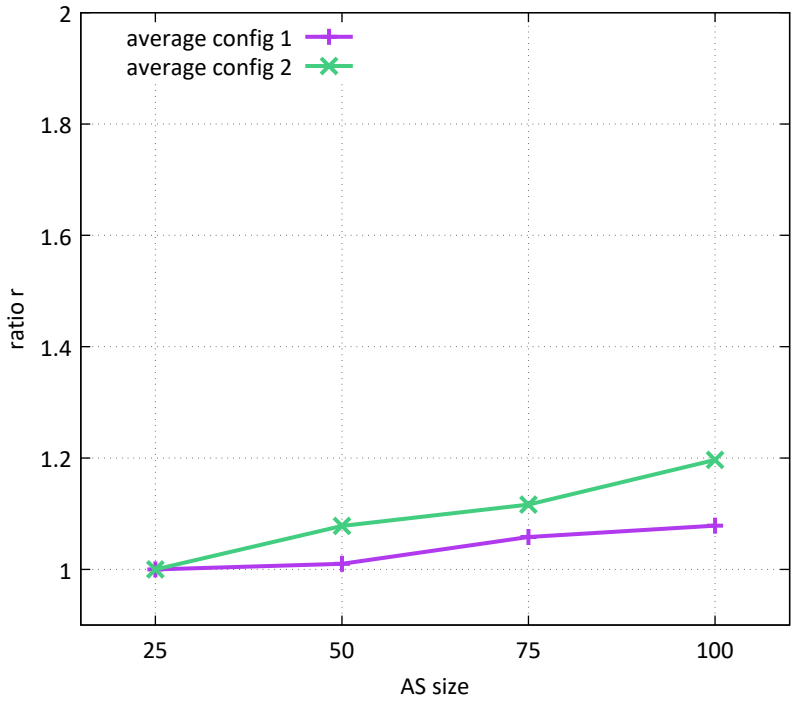


Figure 6.5: Average Ratio of Greedy and Optimal Results

costs would be equal to the optimal costs. A value less than 1 would refute the ILP as an optimal solution. The x-axis shows the AS sizes used for the evaluation and the y-axis the ratio r both on a linear scale. It can be seen that for small networks of 25 nodes, all tests produced the same results regardless of the configuration used. Also, it is visible that with an increasing AS size the average ratio increases slightly from 1 for an AS size of 25 up to 1.2 for an AS size of 100 with the configuration 2 monotonic. The configuration 2 with the higher capacities has a higher average ratio r for all AS sizes except the AS size of 25 where it is equally optimal compared to the configuration 1.

In Fig. 6.6, the maximal deviation from the optimum is visualized. Similar to the distribution of the average ratio t, the maximum of configuration 2 is always higher than configuration 1 except for the AS size of 25. The overall maximal deviation is $\frac{5}{3}$ for an AS with the size of 100 nodes and the configuration 2. In contrast to the average, there is no monotonic growth of the deviation by the AS size.

6.3 MaxiNet Performance

The deployment of the MaxiNet experiments generated by EmuFog is another crucial step for the end-to-end workflow. Running times of EmuFog have been examined previously

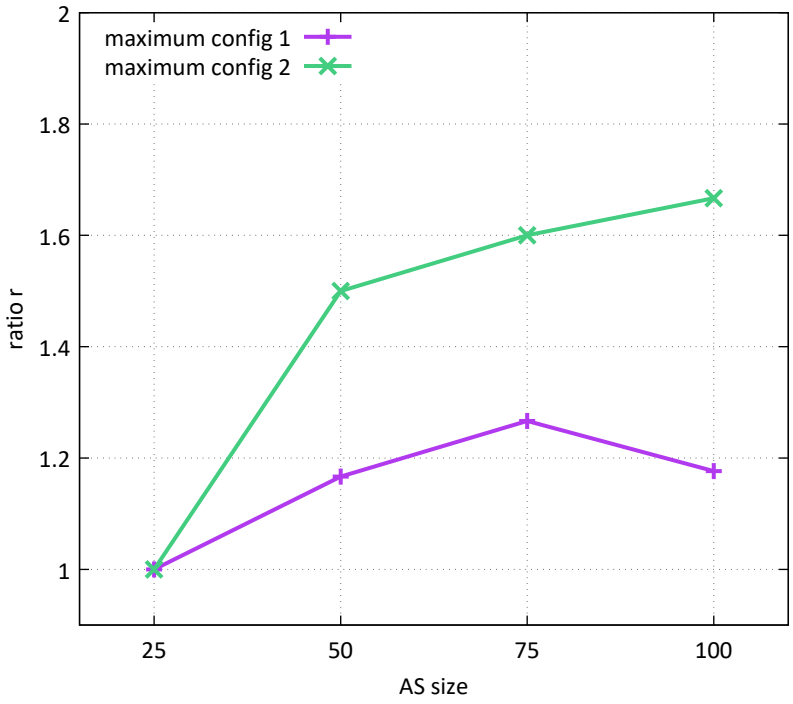


Figure 6.6: Maximum Ratio of Greedy and Optimal Results

and hence the deployment time of the generated experiments needs to be evaluated too. In addition to the performance evaluations performed by Wette et al. [WDS14], this work evaluates the startup time of experiments generated by EmuFog in a distributed environment. Therefore, three Ubuntu 16.04 LTS servers from the Amazon Web Services (AWS) platform got connected setup with MaxiNet. Each server was configured with a *t2.micro* instance that consists of one shared CPU and 1GB of RAM. Fig. 6.7 depicts the result of all measurements. Each AS size got measured by five runs and aggregated to the median. A topology of an AS contains Docker container and basic switches as generated by EmuFog. It is visible that the deployment time increases monotonic by the AS sizes. The running times measured can only provide a tendency of the course as they depend on the environment. A different server or a different number of servers might lead to other results. Thereby the shared CPU and limited RAM affect the deployment time the most. A server with higher computation power can deploy its local Mininet instance faster.

In addition to the topologies generated by EmuFog, this section evaluates the performance of MaxiNet in combination with the startup of many Docker containers. For this purpose, special ring topologies are used. Such a ring consists of an arbitrary number of switches connected to a ring where each switch is connected to five Docker containers. This way the number of switches necessary to connect the topology is reduced to a reasonable overhead compared to the number of Docker containers. For each topology of

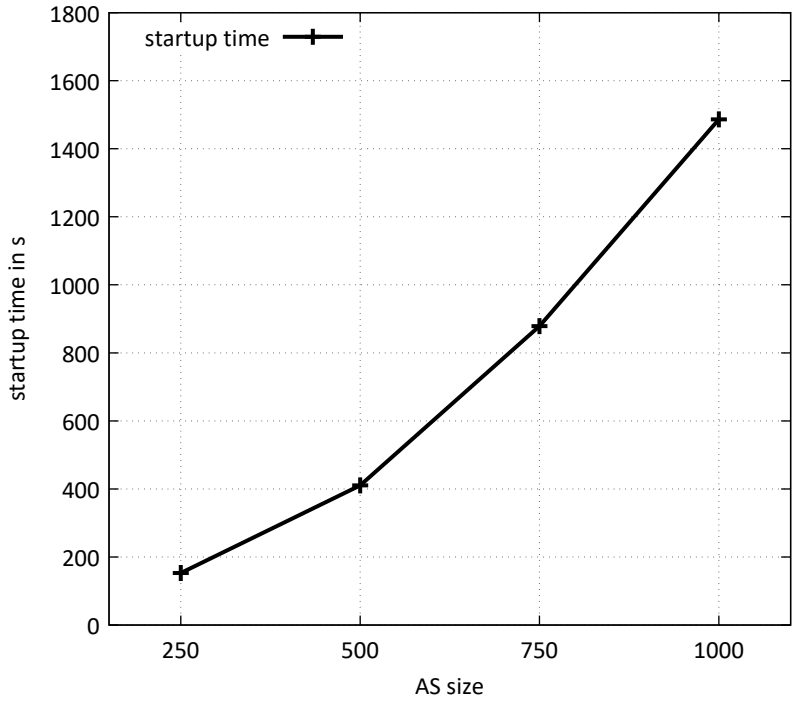


Figure 6.7: Startup Time of MaxiNet Experiments

n containers, there are another $\frac{n}{5}$ switches on top. For the evaluation all containers, use the Ubuntu 16.04 LTS Docker image[3] which is locally present on all servers before the measurement. Due to the limited RAM available on a t2.micro instance in combination with the necessary RAM to start up a Docker container this environment uses six instead of three servers. Exactly as in the previous measure, each AS size got measured five times and aggregated to the median. The results of the measurement are shown in Fig. 6.8. Similar to the performance of general network topologies the growth is monotonic. Comparing the results for 250 nodes with the results in Fig. 6.7 it is visible that the Docker topology starts faster. But this is only due to the fact that there are less links in this topology to initialize.

[3]Docker Container Ubuntu 16.04 is available at https://store.docker.com/images/ubuntu

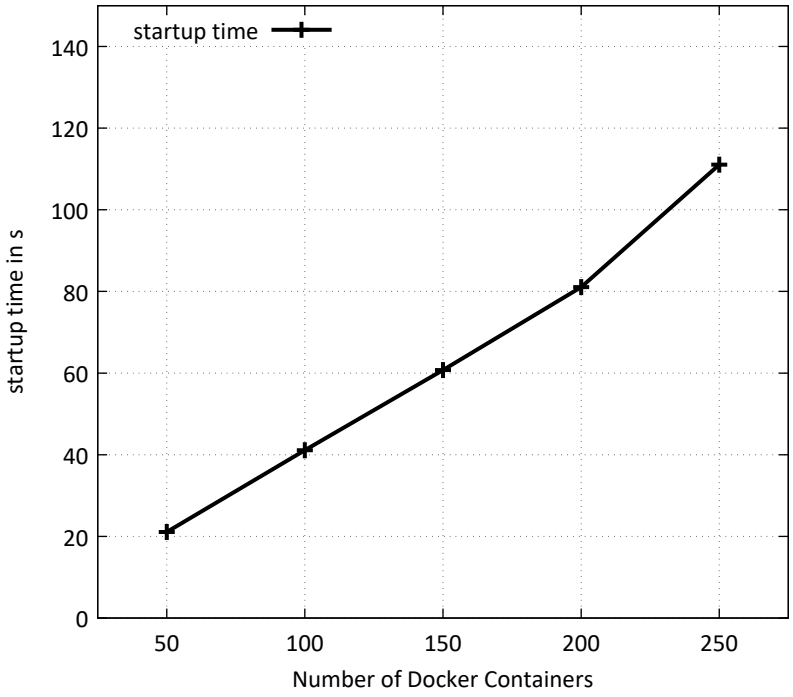


Figure 6.8: Startup Time of MaxiNet Experiments with Docker Containers

7 Conclusion

With EmuFog, this work proposed a new approach for testing Fog Computing using emulation. EmuFog is able to read artificial network topologies by the BRITE network generator, real world topologies measured by Caida, and can easily be extended for other data formats. An algorithm to identify the edge of the network was shown using a combination of connection, degree, and partition characteristics. Based on the identified edge of the network, a second algorithm to place fog nodes was presented. This algorithm is a heuristic to place fog nodes with their respective types cost optimal within a given range defined by a given threshold.

EmuFog was implemented and published open source on the GitHub platform licensed under the MIT license. It is extensible for future use, platform-independent with the Java programming language, and flexible for different use cases. To run generated experiments, an additional network emulator, like MaxiNet or CORE, is required. Currently, an export function for the MaxiNet format is implemented, but for further emulation tools, the respective output format can be expanded easily.

The evaluation presented measured the running time of the previously mentioned algorithms to identify the edge and the fog placement. Thereby, it could also be shown that in the evaluated test topologies, the ratio of the fog placement result to the optimal result is on average less than 1.2 and in the worst case $\frac{5}{3}$. For the final experiments generated by EmuFog, the deployment time using MaxiNet has been evaluated as well as the deployment of Docker containers with MaxiNet.

Although this work provides evaluations of EmuFog and MaxiNet, there is no evaluation of a Fog Computing application tested with EmuFog. This final step should be evaluated in future work for evidence of the end-to-end use case. There is a variety of possible use cases like stream processing, augmented/virtual reality, connected cars, etc. that could be evaluated using EmuFog in combination with MaxiNet.

Currently, it is possible to use Docker container containing the Fog Computing application to evaluate in EmuFog. But it might be useful to pass additional information to emulated nodes via environment variables. Such information could be specified in the settings file to be written dynamically to the output file.

A Autonomous Systems in Detail

Nodes	Edges	Backbone Nodes	Edge Nodes
10	17	4	6
10	27	9	1
10	23	8	2
10	20	8	2
10	23	9	1
100	602	57	43
100	660	71	29
100	657	74	26
100	593	65	35
100	487	57	43
1,000	7,217	592	408
1,000	7,896	596	404
1,000	7,279	635	365
1,000	8,095	604	396
1,000	7,824	593	407
10,000	78,641	5,717	4,273
10,000	78,837	5,739	4,261
10,000	79,157	5,690	4,310
10,000	78,627	5,718	4,282
10,000	79,330	5,855	4,145

Table A.1: The Autonomous Systems of the BRITE Dataset

AS Identifier	Nodes	Edges	Backbone Nodes	Edge Nodes
7486	10	8	9	1
7909	10	9	2	8
9420	10	7	8	2
9587	11	8	9	2
9890	10	6	8	2
60227	98	99	6	92
60672	102	95	8	94
61806	100	99	1	99
6782	95	72	76	19
34660	96	94	13	83
23791	1,002	1,021	32	970
34974	1,007	951	55	952
42730	1,005	999	109	896
39216	1,011	926	19	992
15366	957	946	34	923
12709	9,861	9,811	28	9,833
3758	10,509	11,014	2,551	7,958
5588	10,335	10,160	1,269	9,066
2110	9,728	9,877	159	9,569
2116	9,909	9,277	903	9,006

Table A.2: The Autonomous Systems of the Caida Dataset

Bibliography

[AB00] R. Albert, A.-L. Barabási. “Topology of Evolving Networks: Local Events and Universality.” In: *Physical Review Letters* 85.24 (24 Dec. 2000), pp. 5234–5237. DOI: 10.1103/physrevlett.85.5234. URL: https://doi.org/10.1103%2Fphysrevlett.85.5234 (cit. on pp. 24, 63, 70).

[ADHK08] J. Ahrenholz, C. Danilov, T. R. Henderson, J. H. Kim. “CORE: A real-time network emulator.” In: *MILCOM 2008 - 2008 IEEE Military Communications Conference*. IEEE, Nov. 2008, pp. 1–7. DOI: 10.1109/milcom.2008.4753614. URL: https://doi.org/10.1109/milcom.2008.4753614 (cit. on p. 29).

[AGA11] J. Ahrenholz, T. Goff, B. Adamson. “Integration of the CORE and EMANE Network Emulators.” In: *2011 - MILCOM 2011 Military Communications Conference*. IEEE, Nov. 2011, pp. 1870–1875. DOI: 10.1109/milcom.2011.6127585. URL: https://doi.org/10.1109/milcom.2011.6127585 (cit. on p. 29).

[AGC17] S. Alonso-Monsalve, F. Garcia-Carballeira, A. Calderon. “Fog computing through public-resource computing and storage.” In: *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*. IEEE, May 2017, pp. 81–87. DOI: 10.1109/fmec.2017.7946412. URL: https://doi.org/10.1109/fmec.2017.7946412 (cit. on pp. 18, 21).

[AH14] M. Aazam, E.-N. Huh. “Fog Computing and Smart Gateway Based Communication for Cloud of Things.” In: *2014 International Conference on Future Internet of Things and Cloud*. IEEE, Aug. 2014, pp. 464–470. DOI: 10.1109/ficloud.2014.83. URL: https://doi.org/10.1109/ficloud.2014.83 (cit. on pp. 18, 21).

[Ahr10] J. Ahrenholz. “Comparison of CORE network emulation platforms.” In: *2010 - MILCOM 2010 MILITARY COMMUNICATIONS CONFERENCE*. Institute of Electrical and Electronics Engineers (IEEE), Oct. 2010, pp. 166–171. DOI: 10.1109/milcom.2010.5680218. URL: https://doi.org/10.1109%2Fmilcom.2010.5680218 (cit. on p. 29).

[BA99] A.-L. Barabási, R. Albert. “Emergence of Scaling in Random Networks.” In: *Science* 286.5439 (Oct. 1999), pp. 509–512. ISSN: 0036-8075. DOI: 10.1126/science.286.5439.509. URL: https://doi.org/10.1126%2Fscience.286.5439.509 (cit. on p. 24).

[BEH+02] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, M. S. Marshall. “GraphML Progress Report Structural Layer Proposal.” In: *Graph Drawing: 9th International Symposium, GD 2001 Vienna, Austria, September 23–26, 2001 Revised Papers*. Ed. by P. Mutzel, M. Jünger, S. Leipert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 501–512. ISBN: 978-3-540-45848-7. DOI: 10.1007/3-540-45848-4_59. URL: https://doi.org/10.1007/3-540-45848-4_59 (cit. on p. 35).

[BMNZ14] F. Bonomi, R. Milito, P. Natarajan, J. Zhu. “Fog Computing: A Platform for Internet of Things and Analytics.” In: *Big Data and Internet of Things: A Roadmap for Smart Environments*. Ed. by N. Bessis, C. Dobre. Cham: Springer International Publishing, Mar. 11, 2014, pp. 169–186. ISBN: 978-3-319-05029-4. DOI: 10.1007/978-3-319-05029-4_7. URL: https://doi.org/10.1007/978-3-319-05029-4_7 (cit. on p. 20).

[BMZA12] F. Bonomi, R. Milito, J. Zhu, S. Addepalli. “Fog Computing and Its Role in the Internet of Things.” In: *Proceedings of the first edition of the MCC workshop on Mobile cloud computing - MCC 12*. MCC 12. Helsinki, Finland: ACM Press, 2012, pp. 13–16. ISBN: 978-1-4503-1519-7. DOI: 10.1145/2342509.2342513. URL: https://doi.org/10.1145%2F2342509.2342513 (cit. on pp. 17, 19).

[BT02] T. Bu, D. Towsley. “On distinguishing between Internet power law topology generators.” In: *Proceedings.Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*. Vol. 2. Institute of Electrical and Electronics Engineers (IEEE), 2002, 638–647 vol.2. DOI: 10.1109/infcom.2002.1019309. URL: https://doi.org/10.1109%2Finfcom.2002.1019309 (cit. on pp. 24, 26).

[CDZ97] K. Calvert, M. Doar, E. Zegura. “Modeling Internet topology.” In: *IEEE Communications Magazine* 35.6 (June 1997), pp. 160–163. ISSN: 0163-6804. DOI: 10.1109/35.587723. URL: https://doi.org/10.1109%2F35.587723 (cit. on pp. 22, 23, 25).

[CLRS09] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, Sept. 11, 2009. ISBN: 978-0-262-03384-8 (cit. on pp. 48, 49).

[DCKM04] F. Dabek, R. Cox, F. Kaashoek, R. Morris. “Vivaldi: A Decentralized Network Coordinate System.” In: *ACM SIGCOMM Computer Communication Review* 34.4 (Oct. 2004), pp. 15–26. ISSN: 0146-4833. DOI: 10.1145/1030194.1015471. URL: https://doi.org/10.1145/1030194.1015471 (cit. on p. 59).

[Dij59] E. W. Dijkstra. “A note on two problems in connexion with graphs.” In: *Numerische Mathematik* 1.1 (Dec. 1959), pp. 269–271. ISSN: 0945-3245. DOI: 10.1007/bf01386390. URL: https://doi.org/10.1007%2Fbf01386390 (cit. on p. 46).

[Doa96] M. Doar. “A better model for generating test networks.” In: *Proceedings of GLOBECOM’96. 1996 IEEE Global Telecommunications Conference*. Institute of Electrical and Electronics Engineers (IEEE), Nov. 1996, pp. 86–93. DOI: 10.1109/glocom.1996.586131. URL: https://doi.org/10.1109%2Fglocom.1996.586131 (cit. on pp. 22, 25).

[FFF99] M. Faloutsos, P. Faloutsos, C. Faloutsos. “On Power-law Relationships of the Internet Topology.” In: *ACM SIGCOMM Computer Communication Review* 29.4 (Oct. 1999), pp. 251–262. ISSN: 0146-4833. DOI: 10.1145/316194.316229. URL: https://doi.org/10.1145%2F316194.316229 (cit. on p. 23).

[GKN+11] A. Gluhak, S. Krco, M. Nati, D. Pfisterer, N. Mitton, T. Razafindralambo. “A survey on facilities for experimental internet of things research.” In: *IEEE Communications Magazine* 49.11 (Nov. 2011), pp. 58–67. ISSN: 0163-6804. DOI: 10.1109/mcom.2011.6069710. URL: https://doi.org/10.1109/MCOM.2011.6069710 (cit. on p. 26).

[GSG02] K. P. Gummadi, S. Saroiu, S. D. Gribble. “King: Estimating Latency Between Arbitrary Internet End Hosts.” In: *Proceedings of the second ACM SIGCOMM Workshop on Internet measurment - IMW ’02*. IMW ’02. Marseille, France: ACM Press, 2002, pp. 5–18. ISBN: 1-58113-603-X. DOI: 10.1145/637201.637203. URL: https://doi.org/10.1145/637201.637203 (cit. on p. 59).

[GVGB17] H. Gupta, A. Vahid Dastjerdi, S. K. Ghosh, R. Buyya. “iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments.” In: *Software: Practice and Experience* 47.9 (June 2017). spe.2509, pp. 1275–1296. ISSN: 1097-024X. DOI: 10.1002/spe.2509. URL: https://doi.org/10.1002/spe.2509 (cit. on p. 21).

[Him97] M. Himsolt. “GML: A portable graph file format.” In: (1997). URL: https://www.researchgate.net/publication/228572038 (cit. on p. 35).

[HLC+14] S. N. Han, G. M. Lee, N. Crespi, K. Heo, N. V. Luong, M. Brut, P. Gatellier. "DPWSim: A simulation toolkit for IoT applications using devices profile for web services." In: *2014 IEEE World Forum on Internet of Things (WF-IoT)*. IEEE, Mar. 2014, pp. 544–547. DOI: 10.1109/wf-iot.2014.6803226. URL: https://doi.org/10.1109/wf-iot.2014.6803226 (cit. on p. 21).

[HPSS03] O. Heckmann, M. Piringer, J. Schmitt, R. Steinmetz. "On Realistic Network Topologies for Simulation." In: *Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research - MoMeTools '03*. MoMeTools '03. Karlsruhe, Germany: Association for Computing Machinery (ACM), 2003, pp. 28–32. ISBN: 1-58113-748-6. DOI: 10.1145/944773.944779. URL: https://doi.org/10.1145%2F944773.944779 (cit. on p. 26).

[KK95] G. Karypis, V. Kumar. *METIS – Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0*. Tech. rep. Jan. 1995. URL: https://www.researchgate.net/publication/246815679 (cit. on p. 28).

[KNF+11] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, M. Roughan. "The Internet Topology Zoo." In: *IEEE Journal on Selected Areas in Communications* 29.9 (Oct. 2011), pp. 1765–1775. ISSN: 0733-8716. DOI: 10.1109/jsac.2011.111002. URL: https://doi.org/10.1109%2Fjsac.2011.111002 (cit. on p. 22).

[LHC03] H. Lim, J. C. Hou, C.-H. Choi. "Constructing Internet Coordinate System Based on Delay Measurement." In: *Proceedings of the 2003 ACM SIGCOMM conference on Internet measurement - IMC '03*. IMC '03. Miami Beach, FL, USA: ACM Press, 2003, pp. 129–142. ISBN: 1-58113-773-7. DOI: 10.1145/948205.948222. URL: https://doi.org/10.1145/948205.948222 (cit. on p. 59).

[LHM10] B. Lantz, B. Heller, N. McKeown. "A Network in a Laptop: Rapid Prototyping for Software-defined Networks." In: *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks - Hotnets '10*. Hotnets-IX. Monterey, California: Association for Computing Machinery (ACM), 2010, 19:1–19:6. ISBN: 978-1-4503-0409-2. DOI: 10.1145/1868447.1868466. URL: https://doi.org/10.1145%2F1868447.1868466 (cit. on p. 27).

[LPD11] E. Lochin, T. Pérennou, L. Dairaine. "When should I use network emulation?" In: *annals of telecommunications - annales des télécommunications* 67.5-6 (July 2011), pp. 247–255. ISSN: 1958-9395. DOI: 10.1007/s12243-011-0268-5. URL: https://doi.org/10.1007%2Fs12243-011-0268-5 (cit. on p. 27).

[MLMB01] A. Medina, A. Lakhina, I. Matta, J. Byers. “BRITE: an approach to universal topology generation.” In: *MASCOTS 2001, Proceedings Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. Institute of Electrical and Electronics Engineers (IEEE), 2001, pp. 346–353. DOI: 10.1109/mascot.2001.948886. URL: https://doi.org/10.1109%2Fmascot.2001.948886 (cit. on p. 24).

[MMB00] A. Medina, I. Matta, J. Byers. “On the Origin of Power Laws in Internet Topologies.” In: *ACM SIGCOMM Computer Communication Review* 30.2 (Apr. 2000), pp. 18–28. ISSN: 0146-4833. DOI: 10.1145/505680.505683. URL: https://doi.org/10.1145%2F505680.505683 (cit. on pp. 23, 24, 26).

[Nor16] A. Nordrum. *Popular Internet of Things Forecast of 50 Billion Devices by 2020 Is Outdated*. Aug. 16, 2016. URL: https://spectrum.ieee.org/tech-talk/telecom/internet/popular-internet-of-things-forecast-of-50-billion-devices-by-2020-is-outdated (visited on 09/14/2017) (cit. on p. 17).

[NZ02] T. Ng, H. Zhang. “Predicting Internet network distance with coordinates-based approaches.” In: *Proceedings.Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*. Vol. 1. IEEE, 2002, 170–179 vol.1. DOI: 10.1109/infcom.2002.1019258. URL: https://doi.org/10.1109/infcom.2002.1019258 (cit. on p. 59).

[Ope17] OpenFog Consortium Architecture Working Group. “OpenFog Architecture Overview.” In: *White Paper, February* (Feb. 2017). URL: https://www.openfogconsortium.org/ra/ (cit. on p. 18).

[PBG+13] G. Z. Papadopoulos, J. Beaudaux, A. Gallais, T. Noel, G. Schreiner. “Adding value to WSN simulation using the IoT-LAB experimental platform.” In: *2013 IEEE 9th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. IEEE, Oct. 2013, pp. 485–490. DOI: 10.1109/wimob.2013.6673403. URL: https://doi.org/10.1109/WiMOB.2013.6673403 (cit. on p. 26).

[QSFB09] B. Quoitin, V. V. den Schrieck, P. Francois, O. Bonaventure. “IGen: Generation of router-level Internet topologies through network design heuristics.” In: *2009 21st International Teletraffic Congress*. Sept. 2009, pp. 1–8. ISBN: 978-1-4244-4744-2. URL: https://ieeexplore.ieee.org/document/5300250/ (cit. on p. 24).

[SCM15] S. Sarkar, S. Chatterjee, S. Misra. “Assessment of the Suitability of Fog Computing in the Context of Internet of Things.” In: *IEEE Transactions on Cloud Computing* PP.99 (2015), pp. 1–1. ISSN: 2168-7161. DOI: 10.1109/tcc.2015.2485206. URL: https://doi.org/10.1109/tcc.2015.2485206 (cit. on p. 18).

[SG16] M. Slabicki, K. Grochla. “Performance evaluation of CoAP, SNMP and NETCONF protocols in fog computing architecture.” In: *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*. IEEE, Apr. 2016, pp. 1315–1319. DOI: 10.1109/noms.2016.7503010. URL: https://doi.org/10.1109/noms.2016.7503010 (cit. on p. 21).

[SMG+14] L. Sanchez, L. Muñoz, J. A. Galache, P. Sotres, J. R. Santana, V. Gutierrez, R. Ramdhany, A. Gluhak, S. Krco, E. Theodoridis, D. Pfisterer. “SmartSantander: IoT experimentation over a smart city testbed.” In: *Computer Networks* 61 (Mar. 2014). Special issue on Future Internet Testbeds – Part I, pp. 217–238. ISSN: 1389-1286. DOI: 10.1016/j.bjp.2013.12.020. URL: https://doi.org/10.1016/j.bjp.2013.12.020 (cit. on p. 26).

[SOE17] C. Sonmez, A. Ozgovde, C. Ersoy. “EdgeCloudSim: An environment for performance evaluation of Edge Computing systems.” In: *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*. IEEE, May 2017, pp. 39–44. DOI: 10.1109/fmec.2017.7946405. URL: https://doi.org/10.1109/fmec.2017.7946405 (cit. on p. 21).

[SPPS08] M. Szymaniak, D. Presotto, G. Pierre, M. van Steen. “Practical large-scale latency estimation.” In: *Computer Networks* 52.7 (May 2008), pp. 1343–1364. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2007.11.022. URL: https://doi.org/10.1016/j.comnet.2007.11.022 (cit. on p. 59).

[Sta16] Statista, Inc. *Global desktop PC shipments from 2010 to 2020*. Jan. 2016. URL: https://www.statista.com/statistics/269044/worldwide-desktop-pc-shipments-forecast/ (visited on 09/14/2017) (cit. on p. 17).

[SXBL06] P. Sharma, Z. Xu, S. Banerjee, S.-J. Lee. “Estimating Network Proximity and Latency.” In: *ACM SIGCOMM Computer Communication Review* 36.3 (July 2006), pp. 39–50. ISSN: 0146-4833. DOI: 10.1145/1140086.1140092. URL: https://doi.org/10.1145/1140086.1140092 (cit. on p. 59).

[TGJ+02] H. Tangmunarunkit, R. Govindan, S. Jamin, S. Shenker, W. Willinger. “Network Topology Generators: Degree-based vs. Structural.” In: *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM ’02. Pittsburgh, Pennsylvania, USA: ACM Press, 2002, pp. 147–159. ISBN: 1-58113-570-X. DOI: 10.1145/633025.633040. URL: https://doi.org/10.1145%2F633025.633040 (cit. on p. 25).

[Wax88] B. Waxman. “Routing of multipoint connections.” In: *IEEE Journal on Selected Areas in Communications* 6.9 (Dec. 1988), pp. 1617–1622. ISSN: 0733-8716. DOI: 10.1109/49.12889. URL: https://doi.org/10.1109%2F49.12889 (cit. on p. 22).

[WCU+15] S. Wang, K. Chan, R. Urgaonkar, T. He, K. K. Leung. “Emulation-based study of dynamic service placement in mobile micro-clouds.” In: *MILCOM 2015 - 2015 IEEE Military Communications Conference*. IEEE, Oct. 2015, pp. 1046–1051. DOI: 10.1109/milcom.2015.7357583. URL: https://doi.org/10.1109/milcom.2015.7357583 (cit. on p. 18).

[WDS14] P. Wette, M. Draxler, A. Schwabe. “MaxiNet: Distributed emulation of software-defined networks.” In: *2014 IFIP Networking Conference*. Institute of Electrical and Electronics Engineers (IEEE), June 2014, pp. 1–9. DOI: 10.1109/ifipnetworking.2014.6857078. URL: https://doi.org/10.1109%2Fifipnetworking.2014.6857078 (cit. on pp. 27, 72).

[WJ02] J. Winick, S. Jamin. *Inet-3.0: Internet topology generator*. Tech. rep. Technical Report CSE-TR-456-02, University of Michigan, 2002. URL: https://www.researchgate.net/publication/2522658 (cit. on p. 24).

[WKR02] A. Winter, B. Kullbach, V. Riediger. “An Overview of the GXL Graph Exchange Language.” In: *Software Visualization: International Seminar Dagstuhl Castle, Germany, May 20–25, 2001 Revised Papers*. Ed. by S. Diehl. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 324–336. ISBN: 978-3-540-45875-3. DOI: 10.1007/3-540-45875-1_25. URL: https://doi.org/10.1007/3-540-45875-1_25 (cit. on p. 35).

[WS98] D. J. Watts, S. H. Strogatz. “Collective dynamics of ’small-world’ networks.” In: *Nature* 393.6684 (June 4, 1998), pp. 440–442. DOI: 10.1038/30918. URL: https://doi.org/10.1038/30918 (cit. on p. 26).

[XMR16] Y. Xu, V. Mahendran, S. Radhakrishnan. “Towards SDN-based fog computing: MQTT broker virtualization for effective and reliable delivery.” In: *2016 8th International Conference on Communication Systems and Networks (COMSNETS)*. IEEE, Jan. 2016, pp. 1–6. DOI: 10.1109/comsnets.2016.7439974. URL: https://doi.org/10.1109/comsnets.2016.7439974 (cit. on pp. 18, 21).

[YLL15] S. Yi, C. Li, Q. Li. “A Survey of Fog Computing: Concepts, Applications and Issues.” In: *Proceedings of the 2015 Workshop on Mobile Big Data - Mobidata ’15*. Mobidata ’15. Hangzhou, China: ACM Press, 2015, pp. 37–42. ISBN: 978-1-4503-3524-9. DOI: 10.1145/2757384.2757397. URL: https://doi.org/10.1145%2F2757384.2757397 (cit. on p. 20).

[ZCP+13] J. Zhu, D. S. Chan, M. S. Prabhu, P. Natarajan, H. Hu, F. Bonomi. “Improving Web Sites Performance Using Edge Servers in Fog Computing Architecture.” In: *2013 IEEE Seventh International Symposium on Service-Oriented System Engineering*. IEEE, Mar. 2013, pp. 320–323. DOI: 10.1109/sose.2013.73. URL: https://doi.org/10.1109/sose.2013.73 (cit. on p. 20).

[ZM04] M. Zec, M. Mikuc. “Operating system support for integrated network emulation in imunes.” In: *Workshop on Operating System and Architectural Support for the on demand IT Infrastructure (1; 2004)*. Jan. 2004. URL: https://www.researchgate.net/publication/313752401 (cit. on p. 29).

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature